Microsoft® Windows™ Programmer's Reference

New for Version 3

MICROSOFT®
WINDOWS™
PROGRAMMER'S
REFERENCE
LIBRARY

Microsoft® Windows™ Programmer's Reference

New for Version 3

Written, edited, and produced by
Microsoft Corporation

Distributed by Microsoft Press

MICROSOFT®
WINDOWS™
PROGRAMMER'S REFERENCE

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

Library of Congress Cataloging-in-Publications Data

Microsoft Windows programmer's reference / Microsoft Corporation.
p. cm. -- (Microsoft Windows programmer's reference library)
Includes index.
ISBN 1-55615-309-0
1. Microsoft Windows (Computer programs) I. Microsoft.
II. Series.
QA76.76.W56M533 1990
005.4'3--dc20 90-6037
CIP

6 7 8 9 FGFG 4 3 2 1

Distributed to the book trade in Canada by Macmillan of Canada, a division of Canada Publishing Corporation.
Distributed to the book trade outside the United States and Canada by Penguin Books Ltd.
Penguin Books Ltd., Harmondsworth, Middlesex, England
Penguin Books Australia Ltd., Ringwood, Victoria, Australia
Penguin Books N.Z. Ltd., 182-190 Wairau Road, Auckland 10, New Zealand

Document No. SY0302a-300-R00-1089

Foreword

The *Microsoft Windows Programmer's Reference Library* is the core documentation for Windows programmers that Microsoft provides with the Microsoft® Windows™ Software Development Kit (SDK). The information in these books is the most accurate and up-to-date information on Windows programming available anywhere. The information represents everything Microsoft knows about programming Windows version 3.0 with Microsoft C (the recommended Windows programming language) and the tools we provide in the SDK.

Certain example programs and tools referred to in this book are available only in the Microsoft Windows SDK or Microsoft C 6.0 Professional Development System. However, if you are not currently programming for Windows, these volumes will still provide an excellent overview of the services that Microsoft Windows and the SDK provide to programmers—*Microsoft Windows: A Guide to Programming* and *Microsoft Windows Programming Tools* in particular—and an introduction to graphical user interface (GUI) programming. It is our hope that once you have "kicked the tires" of the Windows SDK by reading these books, you'll be thoroughly convinced—and already prepared—to begin Windows programming the Microsoft way.

Then as you continue to explore the Windows programming environment, *Microsoft Windows Programmer's Reference* will answer many of your programming questions. The book provides information on each Windows application programming interface (API) and describes its calls and services. For many Windows programmers, this book is the most frequently "thumbed," dog-eared, and marked-up volume in the set.

The Microsoft Windows Software Development Kit is available from your Microsoft product dealer. For further information on the Windows SDK or to obtain the name of your nearest Microsoft dealer, call the Microsoft Information Center at 1-800-426-9400.

The Windows Software Development Kit

The Windows high-level application programming interface consists of the functions, messages, data structures, data types, and files you need to develop applications that unleash the full capabilities of personal computers using Intel® 286 and 386™ processors. The API's device independence ensures compatibility with a broad array of displays, printers, and other devices, allowing you to concentrate on your applications and their features and implementation. Development tasks are handled automatically, and advanced tools enable you to design icons, dialog boxes, fonts, menus, and other interface elements.

Here are some of the new or improved features:

- Improved and comprehensive *Guide to Programming*, *Advanced Interface Design Guide*, *Reference*, and *Tools* manuals.
- More source-code examples for hands-on learning.
- Improved tools for editing visual resources.
- New online help-engine facility so you can include a Help system with your applications.
- The Microsoft CodeView® for Windows debugger—the powerful yet easy-to-use source-code debugger for any Windows application.
- New code-execution profiler and segment-swapping analysis facility.

Take advantage of the success of the Microsoft Windows environment—use the Microsoft Windows Software Development Kit to develop powerful, feature-rich graphical applications.

Other Recommended Reading

The following books are recommended for efficient Windows programming and are available from Microsoft Press®:

- *Programming Windows*. Charles Petzold. 862 pages, softcover. An updated second edition will be available in October 1990.
- *Windows: Programmer's Problem Solver*. Richard Wilton. 400 pages, softcover. Available November 1990.
- *Microsoft C Run-Time Library Reference*. Covers version 6. Microsoft Corporation. 852 pages, softcover.

Table of Contents

Introduction

Volume 1

PART 1 Windows Functions

Chapter 1 Window Manager Interface Functions

Chapter 4 Functions Directory

PART 2 Windows Messages

Chapter 5 Messages Overview

Chapter 6 Messages Directory

Volume 2

PART 3 General Reference

Chapter 7 Data Types and Structures

Chapter 8 Resource Script Statements

Chapter 9 File Formats

Appendixes

Tables

Introduction

Application Programming Interface

This manual describes the application programming interface (API) of the Microsoft ® Windows ™ presentation manager. The API contains the functions, messages, data structures, data types, statements, and files that application developers use to create programs that run with Windows.

The API can be thought of as a set of tools which, when properly used, creates a Windows application that is portable across a variety of computers.

Windows Features

A Windows application can take advantage of a number of features provided by the API. These features include the following:

- Shared display, memory, keyboard, mouse, and system timer
- Data interchange with other applications
- Device-independent graphics
- Multitasking
- Dynamic linking

Windows allows applications, running simultaneously on the system, to share hardware resources; application developers do not need to write specific code to accomplish this complex task.

The clipboard, another Windows feature, acts as a place for data interchange between applications. The information sent between applications can be in the form of text, bitmaps, or graphic operations. Windows provides a number of functions and messages that regulate the transmission of information with the clipboard. These functions and the corresponding messages are part of the window manager interface, one of several libraries in the API.

Windows contains functions that an application can use for device-independent graphic operations. These functions create output that is compatible with raster displays and printers of varying resolution, as well as with a number of vector devices (plotters). These functions are part of the graphics device interface (GDI), the second of the API libraries.

Windows provides multitasking, which means that several applications can run simultaneously. The functions that affect multitasking and memory management in general are part of the system services interface, the third API library.

Because of the memory limitations imposed by DOS, it is important to keep applications as compact as possible. Windows accomplishes this compaction through dynamic linking and the use of discardable code, which allows an application to load and execute a subset of the library of functions at run time. Only a single copy of a library is necessary, no matter how many applications access it.

Window Manager Interface

The window manager interface contains the functions that create, move, and alter a window, the most basic element in a Windows application. A window is a rectangular region that contains graphic representations of user input, input options, and system output.

Windows is a menu-driven environment; menus are the principal means of presenting options to a user from within an application. The functions that create menus, alter their contents, and obtain the status of menu items are also part of the window manager interface.

The window manager interface also contains functions that create system output. An example of this output is the dialog box that applications use to request user input and to display information.

The window manager interface also contains messages and the functions that process them. A message is a special data structure that contains information about changes within an application. These changes include keyboard, mouse, and timer events, as well as requests for information or actions that an application should carry out.

Window Manager Interface Function Groups

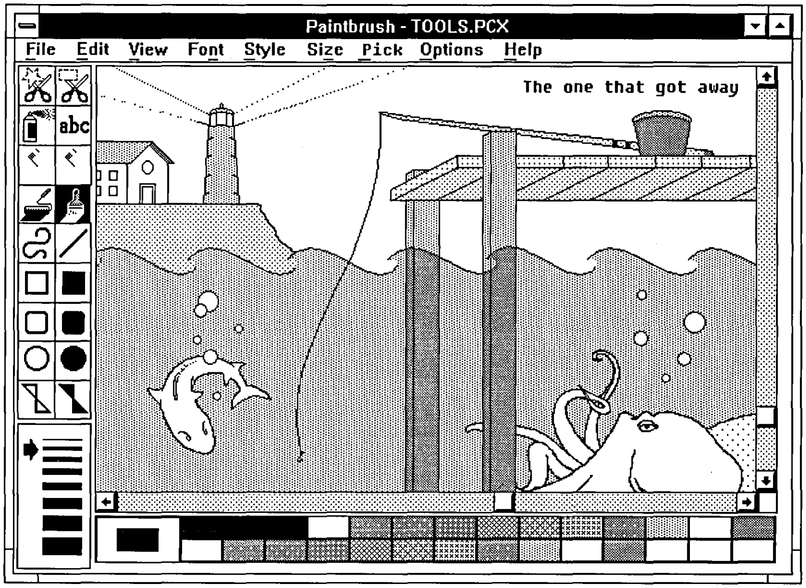
The following list describes the function groups found in the window manager interface:

- Message functions
- Information functions
- Window-creation functions
- System functions
- Display and movement functions
- Clipboard functions
- Error functions

- Input functions
- Caret functions
- Hardware functions
- Cursor functions
- Painting functions
- Hook functions
- Dialog functions
- Property functions
- Scrolling functions
- Rectangle functions
- Menu functions

Graphics Device Interface

The graphics device interface (GDI) contains the functions that perform device-independent graphic operations within a Windows application. These functions create a wide variety of line, text, and bitmap output on a number of different output devices. GDI allows an application to create pens, brushes, fonts, and bitmaps for specific output operations. The following figure shows a sample of text, line, and bitmap output from Microsoft Windows Paintbrush™, an application that was created with GDI functions:



Text, Line and Bitmap Output

Graphics Device Interface Function Groups

The following list describes the function groups found in GDI:

- Device-context functions
- Ellipse and polygon functions
- Drawing-tool functions
- Bitmap functions
- Drawing-attribute functions
- Text functions
- Mapping functions
- Font functions
- Coordinate functions
- Metafile functions
- Region functions
- Printer-escape functions

- Clipping functions
- Environment functions
- Line-output functions
- System functions

System Services Interface

The system services interface contains the functions that access code and data in modules, allocate and manage memory (both local and global), manage tasks, load program resources, translate strings from one character set to another, alter the Windows initialization file, assist in system debugging, carry out communications through the system's I/O ports, create and open files, and create sounds using the system's sound generator.

System Services Interface Function Groups

The following list describes the function groups found in the system services interface:

- Module-management functions
- Initialization-file functions
- Memory-management functions
- Communication functions
- Task functions
- Sound functions
- Resource-management functions
- Utility functions
- String-translation functions
- File I/O functions
- Atom-management functions
- System functions

Naming Conventions

Many Windows functions have been named with a verb-noun model to help you remember and become familiar with the function. The function name indicates both what the function does (verb) and the target of its action (noun). All function names begin with an uppercase letter. If the name is composed of several words, each word begins with an uppercase letter and all words are adjoined (no spaces or underscore characters separate the words). Some examples of function names are shown below:

- **CreateWindow**
- **RegisterClass**
- **SetMapMode**

Parameter Names

Most parameters and local variables have a lowercase prefix that indicates the general type of the parameter, followed by one or more words that describe the content of the parameter. The standard prefixes used in parameter and variable names are defined below:

Prefix	Meaning
b	Boolean (a nonzero value means true, zero means false)
c	Character (a one-byte value)
dw	Long (32-bit) unsigned integer
f	Bit flags packed into a 16-bit integer
h	16-bit handle
l	Long (32-bit) integer
lp	Long (32-bit) pointer
n	Short (16-bit) integer
p	Short (16-bit) pointer
pt	*x*- and *y*-coordinates packed into an unsigned 32-bit integer
rgb	RGB color value packed into a 32-bit integer
w	Short (16-bit) unsigned integer

If no lowercase prefix is given, the parameter is a short integer whose name is descriptive.

Some examples of parameter and variable names are shown as follows:

bIconic	*ptXY*
fAction	*rgbColor*
hWnd	*Height*
lpString	*X*
nBytes	*Width*
pMsg	*Y*

Windows Calling Convention

Windows uses the same calling convention used by Microsoft Pascal. Throughout this manual, this calling convention will be referred to as the Pascal calling convention. The Pascal calling convention entails the following:

- Parameters are pushed onto the stack in the order in which they appear in the function call.
- The code that restores the stack is part of the called function (rather than the calling function).

This convention differs from the calling convention used in other languages, such as C. In C, parameters are pushed onto the stack in reverse order, and the calling function is responsible for restoring the stack.

When developing Windows applications in a language that does not ordinarily use the Pascal calling convention, such as C, you must ensure that the Pascal calling convention is used for any function that is called by Windows. In C, this requires the use of the **PASCAL** key word when the function is declared.

Manual Overview

This manual gives the Windows-application developer general as well as detailed information about Windows functions, messages, data types, resource-compiler statements, assembly-language macros, and file formats. It does not attempt to explain how to create a Windows application. Rather, this manual provides detailed descriptions of each component of the Windows API for readers who already have a basic understanding of Windows programming.

This manual is divided into two volumes. The following sections describe the purpose and contents of each volume.

Volume 1

Volume 1 contains reference information describing the Windows functions and messages. It is made up of six chapters:

Chapter 1, "Window Manager Interface Functions," categorizes window-manager functions into their related groups and briefly describes individual functions. This chapter also supplies additional information about particular function groups, including definitions of new terms and descriptions of models that are unique to Windows. This chapter is designed to assist the application developer who is new to Windows or who has questions about a particular group of Windows functions.

Chapter 2, "Graphics Device Interface Functions," categorizes the functions that perform device-independent graphics operations in the Windows environment, provides brief descriptions of the functions, and explains the most important features of the Windows graphics interface.

Chapter 3, "System Services Interface Functions," categorizes the various utility functions that perform services not directly related to managing a window or producing graphical output.

Chapter 4, "Functions Directory," contains an alphabetical list of Windows functions. The documentation for each function gives the syntax, states the function's purpose, lists its input parameters, and describes its return value. For some functions, additional information the developer needs in order to use those functions is given.

Chapter 5, "Messages Overview," categorizes messages into their related groups and briefly describes individual messages. This chapter also supplies additional information about particular message groups, including definitions of new terms and descriptions of models that are unique to Windows. This chapter is designed to assist the application developer who is new to Windows or who has questions about a particular group of Windows messages.

Chapter 6, "Messages Directory," contains an alphabetical list of Windows messages. The documentation for each message states the message's purpose, lists its input parameters, and describes its return value (if one exists). For some messages, additional information the developer needs in order to use those messages is given.

Volume 2

Volume 2 contains reference material for other components of the Windows API. It contains nine chapters and five appendixes:

Chapter 7, "Data Types and Structures," contains a table of data types and an alphabetical list of structures found in Windows.

Chapter 8, "Resource Script Statements," describes the statements that define resources which the Resource Compiler adds to an application's executable file. The statements are arranged according to functional groups.

Chapter 9, "File Formats," describes the formats of five types of files: bitmap files, icon resource files, cursor resource files, clipboard files, and metafiles. Each description gives the general file structure and information about specific parts of the file.

Chapter 10, "Module-Definition Statements," describes the statements contained in the module-definition file that defines the application's contents and system requirements for the **LINK** program.

Chapter 11, "Binary and Ternary Raster-Operation Codes," describes the raster operations used for line output and those used for bitmap output.

Chapter 12, "Printer Escapes," lists the printer escapes that are available in Windows.

Chapter 13, "Assembly-Language Macros Overview," categorizes and briefly describes the Windows assembly-language macros which provide a simplified interface to the function and segment conventions of high-level languages.

Chapter 14, "Assembly-Language Macros Directory," lists the assembly-language macros alphabetically and, for each macro, provides a detailed description and one or more examples of how to use it in a program.

Chapter 15, "Windows DDE Protocol Definition," contains an alphabetical listing and description of the Windows messages which comprise the Windows Dynamic Data Exchange protocol.

Appendix A, "Virtual-Key Codes," lists the symbolic names and hexadecimal values of Windows virtual-key codes and includes a brief description of each key.

Appendix B, "RC Diagnostic Messages," contains a listing of Resource Compiler error messages and provides a brief description of each message.

Appendix C, "Windows Debugging Messages," contains a listing of Windows debugging messages and provides a brief description of each message.

Appendix D, "Character Tables," shows the layout of the ANSI character set and the IBM PC Extended Character set.

Appendix E, "32-Bit Memory Management DLL," describes how to implement a 32-bit flat memory model for your application.

Document Conventions

Throughout this manual, the term "DOS" refers to both MS-DOS® and PC-DOS, except when noting features that are unique to one or the other.

The following document conventions are used throughout this manual:

Convention	Description of Convention
Bold text	Bold letters indicate a specific term or punctuation mark intended to be used literally: language key words or functions (such as **EXETYPE** or **CreateWindow**), DOS commands, and command-line options (such as **/Zi**). You must type these terms and punctuation marks exactly as shown. However, the use of uppercase or lowercase letters is not always significant. For instance, you can invoke the linker by typing either **LINK**, **link**, or **Link** at the DOS prompt.
( )	In syntax statements, parentheses enclose one or more parameters that you pass to a function.
Italic text	Words in italics indicate a placeholder; you are expected to provide the actual value. For example, the following syntax for the **SetCursorPos** function indicates that you must substitute values for the *X* and *Y* coordinates, separated by a comma: **SetCursorPos**(*X*, *Y*)
`Monospaced type`	Code examples are displayed in a nonproportional typeface.

`BEGIN`
.
.
.
`END`
	Vertical ellipses in program examples indicate that a portion of the program is omitted.

. . .	Ellipses following an item indicate that more items having the same form may appear. In the following example, the horizontal ellipses indicate that you can specify more than one *breakaddress* for the **g** command: **g** [[=*startaddress*]] [[*breakaddress*]]...

[[ ]]	Double brackets enclose optional fields or parameters in command lines and syntax statements. In the following example, *option* and *executable-file* are optional parameters of the **RC** command:
	RC [[*option*]] *filename* [[*executable-file*]]
|	A vertical bar indicates that you may enter one of the entries shown on either side of the bar. The following command-line syntax illustrates the use of a vertical bar:
	DB [[*address* | *range*]]
	The bar indicates that following the **Dump Bytes** command (**DB**), you can specify either an *address* or a *range*.
" "	Quotation marks set off terms defined in the text.
{ }	Curly braces indicate that you must specify one of the enclosed items.
SMALL CAPITAL LETTERS	Small capital letters indicate the names of keys and key sequences, such as:
	ALT + SPACEBAR
$\boxed{3.0}$	A box containing a Microsoft Windows version number indicates that a function, message, or data structure is compatible only with the specified version and later versions.

Microsoft Windows Software Development Kit Documentation Set

Throughout this documentation set, "SDK" refers specifically to the Microsoft Windows Software Development Kit and its contents. The SDK includes the following manuals:

Title	Contents
Installation and Update Guide	Provides an orientation to the SDK, explains how to install the SDK software, and highlights the changes for version 3.0.
Guide to Programming	Explains how to write Windows applications, and provides sample applications that you can use as templates for writing your own programs. The *Guide to Programming* also addresses some more advanced Windows programming topics.
Tools	Explains how to use the software-development tools you'll need to build Windows applications, such as debuggers and specialized SDK editors.
Reference	Is a comprehensive guide to all the details of the Microsoft Windows application program interface (API). The *Reference* lists in alphabetical order all the current functions, messages, and data structures of the API, and provides extensive overviews on how to use the API.
System Application Architecture, Common User Access: Advanced Interface Design Guide	Provides guidelines and recommendations for writing programs that look and act consistently with other Microsoft Windows applications.

Volume 1

Part 1 Windows Functions

Part 1 describes the functions which are the core of the Windows application programmer interface (API). You use these functions as part of a C- or assembly-language program to create an application that takes advantage of Windows' user-interface, graphics and multitasking capabilities.

CHAPTERS

Chapter 1

Window Manager Interface Functions

This chapter describes the Microsoft Windows functions that process messages, create, move, or alter a window, or create system output. These functions constitute the window manager interface.

This chapter describes the following topics:

- Message functions
- Window-creation functions
- Display and movement functions
- Input functions
- Hardware functions
- Painting functions
- Dialog-box functions
- Scrolling functions
- Menu functions
- Information functions
- System functions
- Clipboard functions
- Error functions
- Caret functions
- Cursor functions
- Hook functions
- Property functions
- Rectangle functions

1.1 Message Functions

Message functions read and process Windows messages in an application's queue. Messages represent a variety of input to a Windows application. A message is a data structure that contains a message identifier and message parameters. The content of the parameters varies with the message type. The following list briefly describes each function:

Function	Description
CallWindowProc	Passes message information to the specified function.
DispatchMessage	Passes a message to a window function of the specified window.
GetMessage	Retrieves a message from the specified range of messages.
GetMessagePos	Returns the position of the mouse at the time the last message was retrieved.
GetMessageTime	Returns the time at which the last message was retrieved.
InSendMessage	Determines whether the current window function is processing a message passed to it through a call to the **SendMessage** function.
PeekMessage	Checks the application queue and places the message appropriately.
PostAppMessage	Posts a message to the application.
PostMessage	Places a message in the application queue.
PostQuitMessage	Posts a WM_QUIT message to the application.
ReplyMessage	Replies to a message.
SendMessage	Sends a message to a window or windows.
SetMessageQueue	Creates a new message queue of a different size.
TranslateAccelerator	Processes keyboard accelerators for menu commands.
TranslateMDISysAccel	Processes multiple document interface (MDI) child window command accelerators.
TranslateMessage	Translates virtual key-stroke messages into character messages.
WaitMessage	Yields control to other applications.

Function	Description
WinMain	Serves as an entry point for execution of a Windows application.

1.1.1 Generating and Processing Messages

Windows generates a message at each input event, such as when the user moves the mouse or presses a keyboard key. Windows collects these input messages in a system-wide queue and then places these messages, as well as timer and paint messages, in an application's queue. The application queues are first-in/first-out queues that belong to individual applications; however, timer and paint messages are held in the queue until the application has processed all other messages. Windows places messages that belong to a specific application in that application's queue. The application then reads the messages by using the **GetMessage** function and dispatches them to the appropriate window function by using the **DispatchMessage** function.

Windows sends some messages directly to an application's window function, without placing them in the application queue. Such messages are called unqueued messages. In general, an unqueued message is any message that affects the window only. The **SendMessage** function sends messages directly to a window.

For example, the **CreateWindow** function directs Windows to send a WM_CREATE message to the window function of the application and to wait until the message has been processed by the window function. Windows sends this message directly to the function and does not place it in the application queue.

Although most messages are generated by Windows, applications can create their own messages and place them in the application queues of other applications.

An application can pull messages from its queue by using the **GetMessage** function. This function searches the application queue for messages and, if a message exists, returns the top message in the application queue. If the application queue is empty, **GetMessage** waits for a message to be placed in the queue. While waiting, **GetMessage** relinquishes control to Windows, allowing other applications to take control and process their own messages.

Once a main function has a message from a queue, it can dispatch the message to a window function by using the **DispatchMessage** function. This function directs Windows to call the window function of the window associated with the message, and then passes the content of the message as function arguments. The window function can then process the message and carry out any requested changes to the window. When the window function returns, Windows returns control to the main function. The main function can then pull the next message from the queue.

NOTE Unless noted otherwise, Windows can send messages in any sequence. An application should not rely on receiving messages in a particular order.

Windows generates a virtual-key message each time the user presses a keyboard key. The virtual-key message contains a virtual-key code that defines which key was pressed, but does not define the character value of that key. To retrieve the character value, the main function must translate the virtual-key message by using the **TranslateMessage** function. This function puts another message with an appropriate character value in the application queue. The message can then be dispatched to a window function.

1.1.2 Translating Messages

In general, a main function should use the **TranslateMessage** function to translate every message, not just virtual-key messages. Although **TranslateMessage** has no effect on other types of messages, it guarantees that any keyboard input is translated correctly.

The following program fragment illustrates the typical loop that a main function uses to pull messages from the queues and dispatch them to window functions:

```
int PASCAL WinMain(hInstance, hPrevInstance, lpCmdLine, nShowCmd)
HANDLE hInstance;
HANDLE hPrevInstance;
LPSTR lpCmdLine;
int nShowCmd;
{
     MSG msg;
      .
      .
      .
    while (GetMessage((LPMSG)&msg, NULL, 0, 0))
    {
        TranslateMessage((LPMSG)&msg);
        DispatchMessage((LPMSG)&msg);
    }
    exit(msg.wParam);
}
```

Applications that use accelerator keys must load an accelerator table from the resource file by using the **LoadAccelerator** function, and then translate keyboard messages into accelerator-key messages by using the **Translate-Accelerator** function. The main loop for applications that use accelerator keys should have the following form:

```
while (GetMessage((LPMSG)&msg, (HWND)NULL, 0, 0))
    {
    if (TranslateAccelerator(hWindow, hAccel, ((LPMSG)&msg) == 0)
     {
```

```
            TranslateMessage((LPMSG)&msg);
            DispatchMessage((LPMSG)&msg);
        }
    }
exit(msg.wParam);
```

The **TranslateAccelerator** function must appear before the standard **TranslateMessage** and **DispatchMessage** functions. Furthermore, since **TranslateAccelerator** automatically dispatches the accelerator message to the appropriate window function, the **TranslateMessage** and **DispatchMessage** functions should not be called if **TranslateAccelerator** returns a nonzero value.

1.1.3 Examining Messages

An application can use the **PeekMessage** function when it checks the queues for messages but does not want to pull the message from the queue. The function returns a nonzero value if a message is in the queue, and lets the application retrieve the message and process it without going through the application's main loop.

Typically, an application uses **PeekMessage** to check periodically for messages when the application is carrying out a lengthy operation, such as processing input and output. For example, this function can be used to check for messages that terminate the operation. **PeekMessage** also gives the application a chance to yield control if no messages are present because **PeekMessage** can yield if no messages are in the queue.

1.1.4 Sending Messages

The **SendMessage** and **PostMessage** functions let applications pass messages to their windows or to the windows of other applications.

The **PostMessage** function directs Windows to post the message by placing it in the application queue. Control returns immediately to the calling application, and any action to be carried out as a result of the message does not occur until the message is read from the queue.

The **SendMessage** function directs Windows to send a message directly to the given window function, bypassing the application queue. Windows does not return control to the calling application until the window function that receives the message processes the message.

When an application transmits a message, it must send the message by calling **SendMessage** if the application relies on the return value of a message. The return value of **SendMessage** is the same as the return value of the function that processed the message. **PostMessage** returns immediately after sending the message, so its return value is only a Boolean value indicating whether the message was successfully sent and so does not indicate how the message was processed.

Windows communicates with applications through window messages. The messages are passed (sent or posted) to an application's window function to let the function process the messages as desired. Although an application's main function may read and dispatch window messages, in most cases only the window function processes them.

1.1.5 Avoiding Message Deadlocks

An application can create a deadlock condition in Windows if it yields control while processing a message sent from another application (or by Windows on behalf of another application) by means of the **SendMessage** function. The application does not have to yield explicitly. Calling any one of the following functions can result in the application yielding control:

- **DialogBox**
- **DialogBoxIndirect**
- **DialogBoxIndirectParam**
- **DialogBoxParam**
- **GetMessage**
- **MessageBox**
- **PeekMessage**
- **Yield**

Normally a task that calls **SendMessage** to send a message to another task will not continue executing until the window procedure that receives the message returns. However, if a task that receives the message yields control, Windows can be placed in a deadlock situation where the sending task needs to execute and process messages but cannot because it is waiting for **SendMessage** to return.

A window function can determine whether a message it receives was sent by **SendMessage** by calling the **InSendMessage** function. Before calling any of the functions listed above while processing a message, the window function should first call **InSendMessage.** If **InSendMessage** returns TRUE, the window function must call the **ReplyMessage** function before calling any function that yields control.

As an alternative, can use a system modal dialog box or message box. Because system modal windows prevent other windows from receiving input focus or messages, an application should use system modal windows only when necessary.

1.2 Window-Creation Functions

Window-creation functions create, destroy, modify, and obtain information about windows. The following list briefly describes each window-creation function:

Function	Description
AdjustWindowRect	Computes the size of a window to fit a given client area.
AdjustWindowRectEx	Computes the size of a window with extended style to fit a given client area.
CreateWindow	Creates overlapped, pop-up, and child windows.
CreateWindowEx	Creates overlapped, pop-up, and child windows with extended styles.
DefDlgProc	Provides default processing for those dialog-box messages that an application does not process.
DefFrameProc	Provides default processing for those multiple document interface (MDI) frame window messages that an application does not process.
DefMDIChildProc	Provides default processing those for MDI child window messages an that application does not process.
DefWindowProc	Provides default processing for those window messages that an application does not process.
DestroyWindow	Destroys a window.
GetClassInfo	Retrieves information about a specified class.
GetClassLong	Retrieves window-class information from a **WNDCLASS** structure.
GetClassName	Retrieves a window-class name.
GetClassWord	Retrieves window-class information from a **WNDCLASS** structure.
GetLastActivePopup	Determines which popup window owned by another window was most recently active.
GetWindowLong	Retrieves information about a window.
GetWindowWord	Retrieves information about a window.
RegisterClass	Registers a window class.

Function	Description
SetClassLong	Replaces information in a **WNDCLASS** structure.
SetClassWord	Replaces information in a **WNDCLASS** structure.
SetWindowLong	Changes a window attribute.
SetWindowWord	Changes a window attribute.
UnregisterClass	Removes a window class from the window-class table.

1.2.1 Window Classes

A window class is a set of attributes that defines how a window looks and behaves. Before an application can create and use a window, it must define and register a window class for that window. An application registers a class by passing values for each element of the class to the **RegisterClass** function. Any number of window classes can be registered. Once a class has been registered, Windows lets the application create any number of windows belonging to that class. The registered class remains available until it is deleted or the application terminates.

Although the complete window class consists of many elements, Windows requires only that an application supply a class name, an address to the window procedure that will process all messages sent to windows belonging to this class, and an instance handle that identifies the application that registered the class. The other elements of the window class define default attributes for windows of the class, such as the shape of the cursor and the content of the menu for the window.

There are three types of window classes. They differ in scope and when they are created and destroyed.

System Global Classes

Windows creates system global classes when it starts. These classes are available for use by all applications at all times. Because Windows creates system global classes on behalf of all applications, an application cannot create or destroy any of these classes. Examples of system global classes include edit-control and list-box control classes.

Application Global Classes

An application or (more likely) a library creates an application global class by specifying the CS_GLOBALCLASS style for the class. Once created, it is globally available to all applications within the system. Most often, a library creates an application global class so that applications which call the library can use the class. Windows destroys an application global class when the application or library that created it terminates. For this reason, it is essential that all applications destroy all windows using that class before the library or application that created the class terminates.

Application Local Classes

An application local class is any window class created by an application for its exclusive use. This is the most common type of class created by an application.

1.2.2 How Windows Locates a Class

When an application creates a window with a specified class, Windows uses the following algorithm to find the class:

1. Windows searches for a local class of the specified name.
2. If Windows does not find a local class with the name, then it searches the application global class list.
3. If Windows does not find the name in the application global class list, then it searches the system global class list.

This procedure is used for all windows created by the application, including windows created on the application's behalf, such as dialog controls. It is possible, then, to override system global classes without affecting other applications.

1.2.3 How Windows Determines the Owner of a Class

Windows determines class ownership from the **hInstance** field of the **WNDCLASS** structure passed to the **RegisterClass** function when the application or library registers the class. For Windows libraries, this *must* be the instance handle of the library. When the application that registered the class terminates or the library that registered the class is unloaded, the class is destroyed. For this reason, all windows using the class must be destroyed before the application or library terminates.

1.2.4 Registering a Window Class

When Windows registers a window class, it copies the attributes into its own memory area. Windows uses the internally stored attributes when an application refers to the window class by name; it is not necessary for the application that originally registered the class to keep the structure available.

1.2.5 Shared Window Classes

Applications must not share registered classes with other applications. Some information in a window class, such as the address of the window function, is specific to a given application and cannot be used by other applications. However, applications can share an application global class. See "Application Global Classes," in Section 1.2.1 for more information.

Although applications must not share registered classes, different instances of the same application can share a registered class. Once a window class has been registered by an application, it is available to all subsequent instances of that application. This means that new instances of an application do not need to, and *should* not, register window classes that have been registered by previous instances.

1.2.6 Predefined Window Classes

Windows provides several predefined window classes. These classes define special control windows that carry out common input tasks that let the user input text, direct scrolling, and select from a list of names. The predefined window classes are available to all applications and can be used any number of times to create any number of these control windows.

1.2.7 Elements of a Window Class

The elements of the window class define the default behavior of the windows created from that class. The application that registers the window class assigns elements to the class by setting appropriate fields in a **WNDCLASS** data structure and passing the structure to the **RegisterClass** function. An application can retrieve information about a given window class with the **GetClassInfo** function.

Table 1.1 shows the window class elements:

Table 1.1 Window Class Elements

Element	Purpose
Class name	Distinguishes the class from other registered classes.
Window-function address	Points to the function that processes all messages that are sent to windows in the class, and defines the behavior of the window.
Instance handle	Identifies the application that registered the class.
Class cursor	Defines the shape of the cursor when the cursor is in a window of the class.

Table 1.1 Window Class Elements *(continued)*

Element	Purpose
Class icon	Defines the shape of the icon Windows displays when a window belonging to the class is closed.
Class background brush	Defines the color and pattern Windows uses to fill the client area when the window is opened or painted.
Class menu	Specifies the default menu used for any window in the class that does not explicitly define a menu.
Class styles	Defines how to update the window after moving or resizing, how to process double-clicks of the mouse, how to allocate space for the display context, and other aspects of the window.
Class extra	Specifies the amount of memory (in bytes) that Windows should reserve at the end of the class data structure.
Window extra	Specifies the amount of memory (in bytes) that Windows should reserve at the end of any window structure an application creates with this class.

The following sections describe the elements of a window class and explain the default values for these elements if no explicit value is given when the class is registered.

Class Name

Every window class needs a class name. The class name distinguishes one class from another. An application assigns a class name to the class by setting the **lpszClassName** field of the **WNDCLASS** structure to the address of a null-terminated string that contains the name.

In the case of an application global class, the class name must be unique to distinguish it from other application global classes. If an application registers another application global class with the name of an existing application global class, the **RegisterClass** function returns FALSE, indicating failure. A conventional method for ensuring this uniqueness is to include the application name in the name of the application global class.

The class name must be unique among all the classes registered by an application. An application cannot register an application local class and an application global class with the same class name.

Window-Function Address

Every class needs a window-function address. The address defines the entry point of the window function that is used to process all messages for windows in the class. Windows passes messages to the function when it wants the window to carry out tasks, such as painting its client area or responding to input from the user. An application assigns a window function address by copying the address to the **lpfnWndProc** field of the **WNDCLASS** structure. The window function must be exported in the module-definition (.DEF) file. See Chapter 10, "Module-Definition Statements," in *Reference, Volume 2*, for more information on exporting functions.

For details about the window function, see Section 1.2.13, "Window Function."

Instance Handle

Every window class needs an instance handle to identify the application that registered the class. As a multitasking system, Windows lets several applications run at the same time, so it needs instance handles to keep track of all applications. Windows assigns a unique handle to each copy of a running application.

Windows passes an instance handle to an application when the application first begins operation. The application assigns this instance handle to the class by copying it to the **hInstance** field of the **WNDCLASS** structure.

Class Cursor

The class cursor defines the shape of the cursor when the cursor is in the client area of a window in the class. Windows automatically sets the cursor to the given shape as soon as the cursor enters the window's client area, and ensures that the cursor keeps that shape while it remains in the client area. To assign a cursor shape to a window class, an application typically loads the shape from the application's resources by using the **LoadCursor** function, and then assigns the returned cursor handle to the **hCursor** field of the **WNDCLASS** structure.

Windows does not require a class cursor. If a class cursor is not defined, Windows assumes that the window will set the cursor shape each time the cursor moves into the window.

Class Icon

The class icon defines the shape of the icon used when the window of the given class is minimized. To assign an icon to a window class, an application typically loads the icon from the application's resources by using the **LoadIcon** function, and then assigns the returned icon handle to the **hIcon** field of the **WNDCLASS** structure.

Windows does not require a class icon. If a class icon is not defined, Windows assumes the application will draw the icon whenever the window is minimized.

In this case, Windows sends appropriate messages to the window procedure, requesting that the icon be painted.

Class Background Brush

A class background brush is the brush used to prepare the client area of a window for subsequent drawing by the application. Windows uses the brush to fill the client area with a solid color or pattern, thereby removing all previous images from that location whether they belonged to the window or not.

To assign a background brush to a class, an application typically creates a brush by using the appropriate functions from GDI, and then assigns the returned brush handle to the **hbrBackground** field of the **WNDCLASS** structure.

Instead of creating a brush, an application can use a standard system color by setting the field to one of the following color values:

- COLOR_ACTIVECAPTION
- COLOR_APPWORKSPACE
- COLOR_BACKGROUND
- COLOR_BTNFACE
- COLOR_BTNSHADOW
- COLOR_BTNTEXT
- COLOR_CAPTIONTEXT
- COLOR_GRAYTEXT
- COLOR_HIGHLIGHT
- COLOR_HIGHLIGHTTEXT
- COLOR_INACTIVECAPTION
- COLOR_MENU
- COLOR_MENUTEXT
- COLOR_SCROLLBAR
- COLOR_WINDOW
- COLOR_WINDOWFRAME
- COLOR_WINDOWTEXT

To use a standard system color, the application must increase the background-color value by one. For example, COLOR_BACKGROUND + 1 is the system background color.

Class Menu

A class menu defines the default menu to be used by the windows in the class if no explicit menu is given when the windows are created. A menu is a list of commands that appears at the top of a window, under the title bar, from which a user can select actions for the application to carry out. To assign a menu to a class, an application sets the **lpszMenuName** field of the **WNDCLASS** structure to the address of a null-terminated string that contains the resource name of the menu. The menu is assumed to be a resource in the given application. Windows automatically loads the menu when it is needed. Note that if the menu resource is identified by an integer and not by a name, the **lpszMenuName** field can be set to that integer value by applying the **MAKEINTRESOURCE** macro before assigning the value.

Windows does not require a class menu. If a menu is not given, Windows assumes that the windows in the class have no menu bars. Even if no class menu is given, an application can still define a menu bar for a window when it creates the window.

Windows does not allow menu bars with child windows. If a menu is given and a child window is created using the class, the menu is ignored.

1.2.8 Class Styles

The class styles define additional elements of the window class. Two or more styles can be combined by using the bitwise OR operator. Table 1.2 lists the class styles:

Table 1.2 Window Class Styles

Style	Description
CS_BYTEALIGNCLIENT	Aligns the window's client area on a byte boundary (in the *x* direction).
CS_BYTEALIGNWINDOW	Aligns the window on a byte boundary (in the *x* direction).
CS_CLASSDC	Allocates one display context to be shared by all windows in the class.
CS_DBLCLKS	Sends double-click messages to the window function.

Table 1.2 Window Class Styles (*continued*)

Style	Description
CS_GLOBALCLASS	Specifies that the window class is an application global class. An application global class is created by an application or library and is available to all applications. The class is destroyed when the application or library that created the class terminates; it is essential, therefore, that all windows created with the application global class be closed before this occurs.
CS_HREDRAW	Requests that the entire client area be redrawn if a movement or adjustment to the size changes the width of the client area.
CS_NOCLOSE	Inhibits the close option on the System menu.
CS_OWNDC	Allocates a unique display context for each window in the class.
CS_PARENTDC	Gives the parent window's display context to the window class.
CS_SAVEBITS	Saves the portion of the screen image that is obscured by a window; Windows uses the saved bitmap to re-create the screen image when the window is removed. Windows displays the bitmap at its original location and does not send WM_PAINT messages to windows which had been obscured by the window if the memory used by the bitmap has not been discarded and if other screen actions have not invalidated the stored image.
CS_VREDRAW	Requests that the entire client area be redrawn if a movement or adjustment to the size changes the height of the client area.

To assign a style to a window class, an application assigns the style value to the **style** field of the **WNDCLASS** structure.

1.2.9 Internal Data Structures

Windows maintains internal data structures for each window class and window. These structures are not directly accessible to applications but can be examined and modified by using the following functions:

- **GetClassInfo**
- **GetClassLong**
- **GetClassName**
- **GetClassWord**
- **GetWindowLong**
- **GetWindowWord**
- **SetClassLong**
- **SetClassWord**
- **SetWindowLong**
- **SetWindowWord**

Section 1.2.10 describes some ways in which a window class or window can be modified.

1.2.10 Window Subclassing

A subclass is a window or set of windows that belong to the same window class, and whose messages are intercepted and processed by another window function (or functions) before being passed to the class window function.

To create the subclass, the **SetWindowLong** function is used to change the window function associated with a particular window, causing Windows to call the new window function instead of the previous one. Any messages not processed by the new window function must be passed to the previous window function by calling the **CallWindowProc** function. This allows Windows to create a chain of window functions. The address of the previous window function can be retrieved by using the **GetWindowLong** function before using **SetWindowLong**.

Similarly, the **SetClassLong** function changes the window function associated with a window class. Any window that is subsequently created with that class will be associated with the replacement window function for that class, as will the window whose handle is passed to **SetClassLong**. Other existing windows that were previously created with the class are not affected, however.

When you subclass a window or class of windows, you must export the replacement window procedure in your application's definition file, and you must create the address of the procedure which you pass to **SetWindowLong** or **SetClassLong** by calling the **MakeProcInstance** function.

NOTE An application should not attempt to create a window subclass for standard Windows controls such as combo boxes and buttons.

1.2.11 Redrawing the Client Area

When a window is moved, Windows automatically copies the contents of the client area to the new location. This saves time because a window does not have to recalculate and redraw the contents of the client area as part of the move. If the window moves and changes size, Windows copies only as much of the previous client area as is needed to fill the new location. If the window increases in size, Windows copies the entire client area and sends a WM_PAINT message to the window to fill in the newly exposed areas. When a window is moved, Windows assumes the contents of the client area remain valid and can be copied without modification to the new location.

For some windows, however, the contents of the client area are not valid after a move, especially if the move includes a change in size. For example, a clock application whose window must always contain the complete image of the clock has to redraw the window anytime the window changes size, *and* has to update the time after the move. To prevent the windows from copying the previous contents of the client area, a window should specify the CS_VREDRAW and CS_HREDRAW styles in the window class.

1.2.12 Class and Private Display Contexts

A display context is a special set of values that applications use for drawing in the client area of their windows. Windows requires a display context for each window on the system display, but allows some flexibility in how that display context is stored and treated by the system.

If no explicit display-context style is given, Windows assumes that each window will use a display context retrieved from a pool of contexts maintained by Windows. In such cases, each window must retrieve and initialize the display context before painting, and then free it after painting.

In order not to retrieve a display context each time it wants to paint in a window, an application can specify the CS_OWNDC style for the window class. This class style directs Windows to create a private display context, that is, to allocate a unique display context for each window in the class. The application need only retrieve the context once, and then use it for all subsequent painting. Although the CS_OWNDC style is convenient, it must be used carefully because each display context occupies approximately 800 bytes of memory in the GDI heap.

By specifying the CS_CLASSDC style, an application can have some of the convenience of a private display context without allocating a separate display context for each window. The CS_CLASSDC style directs Windows to create a single class display context, that is, one display context to be shared by all windows in the class. An application need only retrieve the display context for a window; then as long as no other window in the class retrieves that display context, the window can continue to use the context.

Similarly, by specifying the CS_PARENTDC style, an application can create child windows that inherit the device context of their parent.

1.2.13 Window Function

A window function processes all messages sent to a window in a given class. Windows sends messages to a window function when it receives input from the user that is intended for the given window, or when it needs information or the procedure to carry out some action on its window, such as painting in the client area.

A window function receives input messages from the keyboard, mouse, and timer. It receives requests for information, such as a request for the window title. It receives reports of changes made to the system by other windows, such as a change to the WIN.INI file. It receives messages that give it an opportunity to modify the standard system response to certain actions, such as an opportunity to adjust a menu before it is displayed. It receives requests to carry out some action on its window or client area, such as a request to update the client area. And a window function receives information about its status in relation to other windows, such as losing access to the keyboard or becoming the active window.

Most of the messages a window function receives are from Windows, but it can also receive messages from other windows, including windows it owns. These messages can be requests for information or notification that a given event has occurred within another window.

A window function continues to receive messages from the system and possibly other windows in the system until it, or the window function of a parent window, or the system destroys the window. Even in the process of being destroyed, the window function receives additional messages that give it the opportunity to carry out any clean-up tasks before terminating. But once the window is destroyed, no more messages are passed to the function for that particular window. If there is more than one window of the class, however, the window function continues to receive messages for the other windows until they, too, are destroyed.

A window function defines how a given window actually behaves; that is, it defines what response the window makes to commands from the user or system. The messages the window function receives from the system contain information that the function knows; for example, the user clicked the scroll bar or selected the Open command in the File menu, or double-clicked in the client area. The window function must examine these messages and determine what action, if any, to take. For example, if the user clicks the scroll bar, the window function may scroll the contents of the client area. Windows provides detailed information about what happens and provides some tools to carry out tasks, such as drawing and scrolling, but the window function must carry out the actual task.

A window function can also choose not to respond to a given message. If it does not respond, the function must give the system the opportunity to respond by passing the message to the **DefWindowProc** function. This function carries out default actions based on the given message and its parameters. Many messages, especially nonclient-area messages, must be processed, so the **DefWindowProc** function is required in all window functions.

A window function also receives messages that are really intended to be processed by the system. These messages, called nonclient-area messages, inform the function either that the user has carried out some action in a nonclient area of the window, such as clicking the title bar, or that some information about the window is required by the system to carry out an action, such as for moving or adjusting the size of the window. Although Windows passes these messages to the window function, the function should pass them to the **DefWindowProc** function and not attempt to process them. In any case, the window procedure must not ignore the message or return without passing it to **DefWindowProc**.

Window Messages

A window message is a set of values that Windows sends to a window function when it requests some action or informs the window of input. Every message consists of four values: a handle that identifies the window, a message identifier, a 16-bit message-specific value, and a 32-bit message-specific value. These values are passed as individual parameters to the window function. The window function then examines the message identifier to determine what response to make and how to interpret the 16- and 32-bit values.

Windows has a wide variety of messages that it or applications can send to a window function. Most messages are sent to a window as a result of a given function being executed or as input from the user.

To send a message to a window procedure, Windows expects the window function to have four parameters and use the Pascal calling convention. The following illustrates the window procedure syntax:

LONG FAR PASCAL *WndProc*(*hWnd*, *wMsg*, *wParam*, *lParam*)
HWND *hWnd*;
WORD *wMsg*;
WORD *wParam*;
DWORD *lParam*;

The *hWnd* parameter identifies the window receiving the message; the *wMsg* parameter is the message identifier; the *wParam* parameter is 16 bits of additional message-specific information; and *lParam* is 32 bits of additional information. The window procedure must return a 32-bit value that indicates the result of message processing. The possible return values depend on the actual message sent.

Windows expects to make an intersegment call to the window function, so the function must be declared with the **FAR** attribute. The window-function name must be exported by including it in an **EXPORTS** statement in the application's module-definition file.

Default Window Function

The **DefWindowProc** function is the default message processor for window functions that do not or cannot process some of the messages sent to them. For most window functions, the **DefWindowProc** function carries out most, if not all, processing of nonclient-area messages. Those are the messages that signify actions to be carried out on parts of the window other than the client area. Table 1.3 lists the messages **DefWindowProc** processes and the default actions for each:

Table 1.3 Default Actions for Messages

Message	Default Action
WM_ACTIVATE	Sets or kills the input focus.
WM_CANCELMODE	Terminates internal processing of standard scroll bar input, terminates internal menu processing, and releases mouse capture.
WM_CLOSE	Calls the **DestroyWindow** function.
WM_CTLCOLOR	Sets the background and text color and returns a handle to the brush used to fill the control background.
WM_ERASEBKGND	Fills the client area with the color and pattern specified by the class brush, if any.
WM_GETTEXT	Copies the window title into a specified buffer.
WM_GETTEXTLENGTH	Returns the length (in characters) of the window title.
WM_ICONERASEBKGND	Fills the icon client area with the background brush of the parent window.
WM_NCACTIVATE	Activates or deactivates the window and draws the icon or title bar to show the new state.
WM_NCCALCSIZE	Computes the size of the client area.
WM_NCCREATE	Initializes standard scroll bars, if any, and sets the default title for the window.
WM_NCDESTROY	Frees any space internally allocated for the window title.

Table 1.3 Default Actions for Messages (*continued*)

Message	Default Action
WM_NCHITTEST	Determines what part of the window the mouse is in.
WM_NCLBUTTONDBLCLK	Tests the given point to determine the location of the mouse and, if necessary, generates additional messages.
WM_NCLBUTTONDOWN	Determines whether the left mouse button was pressed while the mouse was in the nonclient area of a window.
WM_NCLBUTTONUP	Tests the given point to determine the location of the mouse and, if necessary, generates additional messages.
WM_NCMOUSEMOVE	Tests the given point to determine the location of the mouse and, if necessary, generates additional messages.
WM_NCPAINT	Paints the nonclient parts of the window.
WM_PAINT	Validates the current update region, but does not paint the region.
WM_PAINTICON	Draws the window class icon when a window is minimized.
WM_QUERYENDSESSION	Returns TRUE.
WM_QUERYOPEN	Returns TRUE.
WM_SETREDRAW	Forces an immediate update of information about the clipping area of the complete window.
WM_SETTEXT	Sets and displays the window title.
WM_SHOWWINDOW	Opens or closes a window.
WM_SYSCHAR	Generates a WM_SYSCOMMAND message for menu input.
WM_SYSCOMMAND	Carries out the requested system command.
WM_SYSKEYDOWN	Examines the given key and generates a WM_SYSCOMMAND message if the key is either TAB or ENTER.

1.2.14 Window Styles

Windows provides several different window styles that can be combined to form different kinds of windows. The styles are used in the **CreateWindow** function when the window is created.

Overlapped Windows

An overlapped window is always a top-level window. In other words, an overlapped window never has a parent window. It has a client area, a border, and a title bar. It can also have a System menu, minimize/maximize boxes, scroll bars, and a menu, if these items are specified when the window is created. For windows used as a main interface, the System menu and minimize/maximize boxes are strongly recommended.

Every overlapped window can have a corresponding icon that Windows displays when the window is minimized. A minimized window is not destroyed. It can be opened again by restoring the icon. An application minimizes a window to save screen space when several windows are open at the same time.

You create an overlapped window by using the WS_OVERLAPPED or WS_OVERLAPPEDWINDOW style with the **CreateWindow** function. An overlapped window created with the WS_OVERLAPPED style always has a caption and a border. The WS_OVERLAPPEDWINDOW style creates an overlapped window with a caption, a thick-frame border, a system menu, and minimize and maximize boxes.

Owned Windows

An owned window is a special type of overlapped window. Every owned window has an owner. This owner must also be an overlapped window. Being owned forces several constraints on a window:

- An owned window will always be "above" its owner when the windows are ordered. Attempting to move the owner above the owned window will cause the owned window to also change position to ensure that it will always be above its owner.
- Windows automatically destroys an owned window when it destroys the window's owner.
- An owned window is hidden when its owner is minimized.

An application creates an owned window by specifying the owner's window handle as the *hWndParent* parameter of the **CreateWindow** function when creating a window that has the WS_OVERLAPPED style.

Dialog boxes are owned windows by default. The function that creates the dialog box receives the handle of the owner window as its *hWndParent* parameter.

Pop-up Windows

Pop-up windows are another special type of overlapped window. The main difference between a pop-up window and an overlapped window is that an overlapped window always has a caption, while the caption bar is optional for a pop-up window. Like overlapped windows, pop-up windows can be owned.

You create a pop-up window by using the WS_POPUP window style with the **CreateWindow** function. A pop-up window can be opened and closed by using the **ShowWindow** function.

Child Windows

A child window is the window style used for windows that are confined to the client area of a parent window. Child windows are typically used to divide the client area of a parent window into different functional areas.

You create a child window by using the WS_CHILD window style with the **CreateWindow** function. A child window can be shown and hidden by using the **ShowWindow** function.

Every child window must have a parent window. The parent window can be an overlapped window, a pop-up window, or even another child window. The parent window relinquishes a portion of its client area to the child window, and the child window receives all input from this area. The window class does not have to be the same for each of the child windows in the parent window. This means an application can fill a parent window with child windows that look different and carry out different tasks.

A child window has a client area, but it does not have any other features unless these are explicitly requested. An application can request a border, title bar, minimize/maximize boxes, and scroll bars for a child window. In most cases, the application designs its own features for the child window.

Although not required, every child window should have a unique integer identifier. The identifier, given in the menu parameter of the **CreateWindow** function in place of a menu, helps identify the child window when its parent window has many other child windows. The child window should use this identifier in any messages it sends to the parent window. This is the way a parent window with several child windows can identify which child window is sending the message.

Windows always positions the child window relative to the upper-left corner of the parent window's client area. The coordinates are always client coordinates. (For information about mapping, see Section 2.5, "Mapping Functions.") If all or part of a child window is moved outside the visible portion of the parent window's client area, the child window is clipped; that is, the portion outside the parent window's client area is not displayed.

A child window is an independent window that receives its own input and other messages. Input intended for a child window goes directly to the child window and is not passed through the parent window. The only exception is if input to the child window has been disabled by the **EnableWindow** function. In this case, Windows passes any input that would have gone to the child window to the parent window instead. This gives the parent window an opportunity to examine the input and enable the child window, if necessary.

Actions that affect the parent window can also affect the child window. The following is a list of actions affecting parent windows that can affect child windows:

Parent Window	Child Window
Shown	Shown after the parent window.
Hidden	Hidden prior to the parent window being closed. A child window can be visible only when the parent window is visible.
Destroyed	Destroyed prior to the parent window being destroyed.
Moved	Moved with the parent window's client area. The child window is responsible for painting after the move.
Increased in size or maximized	Paints any portions of the parent window that have been exposed as a result of the increased size of the client area.

Windows does not automatically clip a child window from the parent window's client area. This means the parent window will draw over the child window if it carries out any drawing in the same location as the child window. Windows does clip the child window from the parent window's client area if the parent window has a WS_CLIPCHILDREN style. If the child window is clipped, the parent window cannot draw over it.

A child window can overlap other child windows in the same client area. Two child windows of the same parent window may draw in each other's client area unless one child window has a WS_CLIPSIBLINGS style. Sibling windows are child windows that share the same parent window. If the application specifies this style for a child window, any portion of that child's sibling window that lies within this window will be clipped.

If a window has either the WS_CLIPCHILDREN or WS_CLIPSIBLINGS style, a slight loss in performance occurs.

1.2.15 Multiple Document Interface Windows

Windows multiple document interface (MDI) provides applications with a standard interface for displaying multiple documents within the same instance of an application. An MDI application creates a frame window which contains a client window in place of its client area. An application creates an MDI client window by calling **CreateWindow** with the class MDICLIENT and passing a **CLIENTCREATESTRUCT** data structure as the function's *lpParam* parameter. This client window in turn can own multiple child windows, each of which displays a separate document. An MDI application controls these child windows by sending messages to its client window.

For more information on the multiple document interface, see the *Guide to Programming*.

1.2.16 Title Bar

The title bar, a rectangle at the top of the window, provides space for the window title or name. An application defines the window title when it creates the window. It can also change this name anytime by using the **SetWindowText** function. If a window has a title bar, Windows lets the user use the mouse to move the window.

1.2.17 System Menu

The System menu, identified by an icon at the left end of the title bar, is a pop-up menu that contains the system commands. The system commands are commands selected by the user to direct Windows to carry out actions on the window, such as moving and closing it.

If a System menu or close box is desired for a window, the WS_SYSMENU and WS_CAPTION window styles must be specified when the window is created.

1.2.18 Scroll Bars

The horizontal and vertical scroll bars, bars on the right and lower sides of a window, let a user scroll the contents of the client area. Windows sends scroll requests to a window as WM_HSCROLL and WM_VSCROLL messages. If the window permits scrolling, the window function must process these messages.

A window can have one or both scroll bars. To create a window with a scroll bar, the application must specify the WS_HSCROLL or WS_VSCROLL window style when the window is created.

1.2.19 Menus

A menu is a list of commands from which the user can select using the mouse or the keyboard. When the user selects an item, Windows sends a corresponding message to the window function to indicate which command was selected. Windows provides two types of menus: menu bars (sometimes called static menus) and pop-up menus.

A menu bar is a horizontal menu that appears at the top of a window and below the title bar, if one exists. Any window except a child window can have a menu bar. If an application does not specify a menu when it creates a window, the window receives the default menu bar (if any) defined by the window class.

Pop-up menus contain a vertical list of items and are often displayed when a user selects a menu-bar item. In turn, a pop-up menu item can display another pop-up menu. Also, a pop-up menu can be "floating." A floating pop-up menu can appear anywhere on the screen designated by the application. An application creates an empty pop-up menu by calling the **CreatePopupMenu** function, and then fills in the menu using the **AppendMenu** and **InsertMenu** functions. It displays the pop-up menu by calling **TrackPopupMenu.**

Individual menu items can be created or modified with the MF_OWNERDRAW style, indicating that the item is an owner-draw item. In this case, the owner of the menu is responsible for drawing all visual aspects of the menu item, including checked, grayed, and highlighted states. When the menu is displayed for the first time, the window that owns the menu receives a WM_MEASUREITEM message. The *lParam* parameter of this message points to a **MEASUREITEMSTRUCT** data structure. The owner then fills in this data structure with the dimensions of the item and returns. Windows uses the information in the data structure to determine the size of the item so that Windows can appropriately detect the user's interaction with the item.

Windows sends the WM_DRAWITEM message whenever the owner of the menu must update the visual appearance of the item. Unlike other owner-draw controls, however, the owner of the menu item does not receive the WM_DELETEITEM message when the menu item is removed from the menu. A top-level menu item cannot be an owner-draw item.

When the application calls **AppendMenu**, **InsertMenu**, or **ModifyMenu** to add an owner-draw menu item to a menu or to change an existing menu item to be an owner-draw menu item, the application can supply a 32-bit value as the *lpNewItem* parameter to the function. The application can use this value to maintain additional data associated with the item. This value is available to the application as the **itemData** field of the structures pointed to by the *lParam* parameter of the WM_MEASUREITEM and WM_DRAWITEM messages. For example, if an application were to draw the text in a menu item using a specific color, the 32-bit value could contain a pointer to a string. The application could then set the text color before drawing the item when it received the WM_DRAWITEM message.

1.2.20 Window State

The window state can be opened or closed (iconic), hidden or visible, and enabled or disabled. The initial state of a window can be set by using the following window styles:

- WS_DISABLED
- WS_MINIMIZE
- WS_MAXIMIZE

- WS_VISIBLE

Windows creates windows that are initially enabled for input, that is, windows that can start receiving input messages immediately. In some cases, an application may need to disable input to a new window. It can disable input by specifying the WS_DISABLED window style.

A new window is not displayed until an application opens it by using the **ShowWindow** function or specifies the WS_VISIBLE window style when it creates the window. For overlapped windows, the WS_ICONIC window style creates a window that is minimized initially.

1.2.21 Life Cycle of a Window

Because the purpose of any window is to let the user enter data or to let the application display information, a window starts its life cycle when the application has a need for input or output. A window continues its life cycle until there is no longer a need for it, or the application is terminated. Some windows, such as the window used for the application's main user interface, last the life of the application. Other windows, such as a window used as a dialog box, may last only a few seconds.

The first step in a window's life cycle is creation. Given a registered window class with a corresponding window function, the application uses the **CreateWindow** function to create the window. This function directs Windows to prepare internal data structures for the window and to return a unique integer value, called a window handle, that the application can use to identify the window in subsequent function calls.

The first message most windows process is WM_CREATE, the window-creation message. Again, the **CreateWindow** function sends this message to inform the window function that it can now perform any initialization, such as allocating memory and preparing data files. The *wParam* parameter is not used, but the *lParam* parameter contains a long pointer to a **CREATESTRUCT** data structure, whose fields correspond to the parameters passed to **CreateWindow**.

Both the WM_CREATE and WM_NCCREATE messages are sent directly to the window function, bypassing the application queue. This means an application will create a window and process the WM_CREATE message before it enters the main program loop.

After a window has been created, it must be opened (displayed) before it can be used. An application can open the window in one of two ways: it can specify the WS_VISIBLE window style in the **CreateWindow** function to open the window immediately after creation, or it can wait until later and call the **ShowWindow** function to open the window. When creating a main window, an application should not specify WS_VISIBLE, but should call **ShowWindow** from the WinMain function with the *nCmdShow* parameter set to the desired value.

When the window is no longer needed or the application is terminated, the window must be destroyed. This is done by using the **DestroyWindow** function. **DestroyWindow** removes the window from the system display and invalidates the window handle. It also sends WM_DESTROY and WM_NCDESTROY messages to the window function.

The WM_DESTROY message is usually the last message a window function processes. This occurs when the **DestroyWindow** function is called or when a WM_CLOSE message is processed by the **DefWindowProc** function. When a window function receives a WM_DESTROY message, it should free any allocated memory and close any open data files.

The window used as the application's main user interface should always be the last window destroyed and should always cause the application to terminate. When this window receives a WM_DESTROY message, it should call the **PostQuitMessage** function. This function copies a WM_QUIT message to the application's message queue as a signal for the application to terminate when the message is read from the queue.

1.3 Display and Movement Functions

Display and movement functions show, hide, move, and obtain information about the number and position of windows on the screen. The following list briefly describes each display and movement function:

Function	Description
ArrangeIconicWindows	Arranges minimized (iconic) child windows.
BeginDeferWindowPos	Initializes memory used by the **DeferWindowPos** function.
BringWindowToTop	Brings a window to the top of a stack of overlapped windows.
CloseWindow	Hides the specified window or minimizes it.
DeferWindowPos	Records positioning information for a window to be moved or resized by the **EndDeferWindowPos** function.
EndDeferWindowPos	Positions or sizes several windows simultaneously based on information recorded by the **DeferWindowPos** function.
GetClientRect	Copies the coordinates of a window's client area.
GetWindowRect	Copies the dimensions of an entire window.
GetWindowText	Copies a window caption into a buffer.

Function	Description
GetWindowTextLength	Returns the length (in characters) of the given window's caption or text.
IsIconic	Specifies whether a window is open or closed (iconic).
IsWindowVisible	Determines whether the given window is visible.
IsZoomed	Determines whether a window is maximized.
MoveWindow	Changes the size and position of a window.
OpenIcon	Opens the specified window.
SetWindowPos	Changes the size, position, and ordering of child or pop-up windows.
SetWindowText	Sets the window caption or text.
ShowOwnedPopups	Shows or hides all pop-up windows.
ShowWindow	Displays or removes the given window.

1.4 Input Functions

Input functions disable input from system devices, take control of the system devices, or define special actions that Windows takes when an application receives input from a system device. (The system devices are the mouse, the keyboard, and the timer.) The following list briefly describes each input function:

Function	Description
EnableWindow	Enables and disables mouse and keyboard input throughout the application.
GetActiveWindow	Returns a handle to the active window.
GetCapture	Returns a handle to the window with the mouse capture.
GetCurrentTime	Retrieves the current Windows time.
GetDoubleClickTime	Retrieves the current double-click time for the mouse.
GetFocus	Retrieves the handle of the window that currently owns the input focus.
GetTickCount	Returns the number of timer ticks recorded since the system was booted.

Function	Description
IsWindowEnabled	Determines whether the specified window is enabled for mouse and keyboard input.
KillTimer	Kills the specified timer event.
ReleaseCapture	Releases mouse input and restores normal input processing.
SetActiveWindow	Makes a window the active window.
SetCapture	Causes mouse input to be sent to a specified window.
SetDoubleClickTime	Sets the double-click time for the mouse.
SetFocus	Assigns the input focus to a specified window.
SetSysModalWindow	Makes the specified window a system modal window.
SetTimer	Creates a system-timer event.
SwapMouseButton	Reverses the meaning of left and right mouse buttons.

1.5 Hardware Functions

Hardware functions alter the state of input devices and obtain state information. Windows uses the mouse and the keyboard as input devices. The following list briefly describes each hardware function:

Function	Description
EnableHardwareInput	Enables or disables mouse and keyboard input throughout the application.
GetAsyncKeyState	Returns interrupt-level information about the key state.
GetInputState	Returns TRUE if there is mouse or keyboard input.
GetKBCodePage	Determines which OEM/ANSI tables are loaded.
GetKeyboardState	Copies an array that contains the state of keyboard keys.
GetKeyNameText	Retrieves a string containing the name of a key from a list maintained by the keyboard driver.
GetKeyState	Retrieves the state of a virtual key.

Function	Description
MapVirtualKey	Accepts a virtual-key code or scan code for a key and returns the corresponding scan code, virtual-key code, or ASCII value.
OemKeyScan	Maps OEM ASCII codes 0 through 0x0FF into the OEM scan codes and shift states.
SetKeyboardState	Sets the state of keyboard keys by altering values in an array.
VkKeyScan	Translates an ANSI character to the corresponding virtual-key code and shift state for the current keyboard.

1.6 Painting Functions

Painting functions prepare a window for painting and carry out some useful general-purpose graphics operations. Although all the paint functions are specifically intended for the system display, some can be used for other output devices. The following list briefly describes each painting function:

Function	Description
BeginPaint	Prepares a window for painting.
DrawFocusRect	Draws a rectangle in the style used to indicate focus.
DrawIcon	Draws an icon.
DrawText	Draws characters of a specified string.
EndPaint	Marks the end of window repainting.
ExcludeUpdateRgn	Prevents drawing within invalid areas of a window.
FillRect	Fills a given rectangle by using the specified brush.
FrameRect	Draws a border for the given rectangle.
GetDC	Retrieves the display context for the client area.
GetUpdateRect	Copies the dimensions of a window region's bounding rectangle.
GetUpdateRgn	Copies a window's update region.
GetWindowDC	Retrieves the display context for an entire window.
GrayString	Writes the characters of a string using gray text.
InvalidateRect	Marks a rectangle for repainting.

Function	Description
InvalidateRgn	Marks a region for repainting.
InvertRect	Inverts the display bits of the specified rectangle.
ReleaseDC	Releases a display context.
UpdateWindow	Notifies the application when parts of a window need redrawing.
ValidateRect	Releases the specified rectangle from repainting.
ValidateRgn	Releases the specified region from repainting.

1.6.1 How Windows Manages the Display

The system display is the principal display device for all applications running with Windows. All applications are free to display some form of output on the system display, but since many applications can run at one time, applications are not entitled to the entire system display. The complete system display must be shared. Windows shares the system display by carefully managing the access that applications have to it. Windows ensures that applications have space to display output but do not draw in the space reserved for other applications.

Windows manages the system display by using the display context type. The display context is a special device context that treats each window as a separate display surface. An application that retrieves a display context for a specific window has complete control of the system display within that window, but cannot access or paint over any part of the display outside the window. With a display context, an application can use GDI painting functions, as well as the output functions described in this section, to draw in the given window.

1.6.2 Display Context Types

There are four types of display contexts: common, class, private, and window. The common, class, and private display contexts permit drawing in the client area of a given window. The window display context permits drawing anywhere in the window. When a window is created, Windows assigns a common, class, or private display context to it, based on the type of display context specified in that window's class style.

Common Display Context

A common display context is the default context for all windows. Windows assigns a common display context to the window if a display-context type is not explicitly specified in the window's class style.

A common display context permits drawing in a window's client area, but it is not immediately available for use by a window. A common display context must be retrieved from a cache of display contexts before a window can carry out any drawing in its client area. The **GetDC** or **BeginPaint** function retrieves the display context and returns a handle to the context. The handle can be used with GDI functions to draw in the client area of the given window. After drawing is complete, the context must be returned to the cache by using the **ReleaseDC** or **EndPaint** function. After the context is released, drawing cannot occur until another display context is retrieved.

When a common display context is retrieved, Windows gives it default selections for pen, brush, font, clipping area, and other attributes. These attributes define the tools currently available to carry out the actual drawing. Table 1.4 lists the default selections for a common display context:

Table 1.4 Defaults for a Display Context

Attribute	Default
Background color	White
Background mode	OPAQUE
Bitmap	No default.
Brush	WHITE_BRUSH
Brush origin	(0,0)
Clipping region	Entire client area with the update region clipped as appropriate. Child and pop-up windows in the client area may also be clipped.
Color palette	DEFAULT_PALETTE
Current pen position	(0,0)
Device origin	Upper-left corner of client area.
Drawing mode	R2_COPYPEN
Font	SYSTEM_FONT (SYSTEM_FIXED_FONT for applications written to run with Windows versions prior to 3.0)
Intercharacter spacing	0
Mapping mode	MM_TEXT
Pen	BLACK_PEN
Polygon-filling mode	ALTERNATE
Relative-absolute flag	ABSOLUTE
Stretching mode	BLACKONWHITE
Text color	Black
Viewport extent	(1,1)
Viewport origin	(0,0)

Table 1.4 Defaults for a Display Context *(continued)*

Attribute	Default
Window extents	(1,1)
Window origin	(0,0)

An application can modify the attributes of the display context by using the selection functions and display-context attribute functions. For example, applications typically change the selected pen, brush, and font.

When a common display context is released, the current selections, such as mapping mode and clipping area, are lost. Windows does not preserve the previous selections of a common display context since these contexts are shared and Windows has no way to guarantee that the next window to use a given common display context will be the last window to use that context. Applications that modify the attributes of a common display context must do so each time another context is retrieved.

Class Display Context

A window has a class display context if the window class specifies the CS_CLASSDC style. A class display context is shared by all windows in a given class. A class display context is not part of the display context cache. Instead, Windows specifically allocates a class context for sole use by the window class.

A class display context must be retrieved before it can be used, but it does not have to be released after use. As long as only one window from the class uses the context, the class display context can be kept and reused. If another window in the class needs to use the context, that window must retrieve it before any drawing occurs. Retrieving the context sets the correct origin and clipping for the new window and ensures that the context will be applied to the correct window. A handle to the class display context can be retrieved by using the **GetDC** or **BeginPaint** function. The **ReleaseDC** and **EndPaint** functions have no effect on the class display context.

A class display context is given the same default selections as a common display context when the first window of the class is created (see Table 1.4, "Defaults for a Display Context"). These selections can be modified at any time. Windows preserves all new selections made for the class display context, except for the clipping region and device origin, which are adjusted for the current window when the context is retrieved. Otherwise, all other attributes remain unchanged. This means a change made by one window applies to all windows that subsequently use the context.

NOTE Changing the mapping mode of a class display context may have an undesirable effect on how a window's background is erased. For more information, see Section 1.6.7, "Window Background," and Section 2.5, "Mapping Functions."

Private Display Context

A window has a private display context if the window class specifies the CS_OWNDC style. A private display context is used exclusively by a given window. A private display context is not part of the display context cache. Instead, Windows specifically allocates the context for sole use by the window.

A private display context needs to be retrieved only once. Thereafter, it can be kept and used any number of times by the window. Windows automatically updates the context to reflect changes to the window, such as moving or sizing. A handle to a private display context can be retrieved by using the **GetDC** or **BeginPaint** function. The **ReleaseDC** and **EndPaint** functions have no effect on the private display context.

A private display context is given the same default selections as a common display context when the window is created (see Table 1.4, "Defaults for a Display Context"). These selections can be modified at any time. Windows preserves any new selections made for the context. New selections, such as clipping region and brush, remain selected until the window specifically makes a change.

NOTE Changing the mapping mode of a private display context may have an undesirable effect on how the window's background is erased. For more information, see Section 1.6.7, "Window Background," and Section 2.5, "Mapping Functions."

Window Display Context

A window display context permits painting anywhere in a window, including the caption bar, menus, and scroll bars. Its origin is the upper-left corner of the window, instead of the upper-left corner of the client area.

The **GetWindowDC** function retrieves a window display context from the same cache as it does common display contexts. Therefore, a window that uses a window display context must release it with the **ReleaseDC** function immediately after drawing.

Windows always sets the current selections of a window display context to the same default selections as a common display context and does not preserve any change the window may have made to these selections (see Table 1.4, "Defaults for a Display Context"). Windows does not allow private or class window display contexts, so CS_OWNDC and CS_CLASSDC class styles have no effect on the window display context.

A window display context is intended to be used for special painting within a window's nonclient area. Since painting in nonclient areas of overlapped windows is not recommended, most applications reserve a display context for designing custom child windows. For example, an application may use the display context to draw a custom border around the window. In such cases, the window usually processes the WM_NCPAINT message instead of passing it on to the **DefWindowProc** function. For applications that do not process WM_NCPAINT messages but still wish to paint in the nonclient area, the **GetSystemMetrics** function can be used to retrieve the dimensions of various parts of the nonclient area, such as the caption bar, menu bar, and scroll bars.

1.6.3 Display-Context Cache

Windows maintains a cache of display contexts that it uses for common and window display contexts. This cache contains five display contexts, which means only five common display contexts can be active at any one time. To prevent more than five from being retrieved, a window that uses a common or window display context must release that context immediately after drawing.

If a window fails to release a common display context, all five display contexts may eventually be active and unavailable for any other window. In such a case, Windows ignores all subsequent requests for a common display context. In the retail version of Windows, the system will appear to be deadlocked, while the debugging version of Windows will undergo a fatal exit, alerting the developer of a problem.

The **ReleaseDC** function releases a display context and returns it to the cache. Class and private display contexts are individually allocated for each class or window; they do not belong to the cache so they do not need to be released after use.

1.6.4 Painting Sequence

Windows carries out many operations to manage the system display that affect the content of the client area. If Windows moves, sizes, or alters the appearance of the display, the change may affect a given window. If so, Windows marks the area changed by the operation as ready for updating and, at the next opportunity, sends a WM_PAINT message to the window so that it can update the window in the update region. If a window paints in its client area, it must call the **BeginPaint** function to retrieve a handle to a display context, must update the changed area as defined by the update region, and finally, must call the **EndPaint** function to complete the operation.

A window is free to paint in its client area at any time, that is, at times other than in response to a WM_PAINT message. The only requirement is that it retrieve a display context for the client area before carrying out any operations.

1.6.5 WM_PAINT Message

The WM_PAINT message is a request from Windows to a given window to update its display. Windows sends a WM_PAINT message to a window whenever it is necessary to repaint a portion of an application's window. When a window receives a WM_PAINT message, it should retrieve the update region by using the **BeginPaint** function, and it should carry out whatever operations are necessary to update that part of the client area.

The **InvalidateRect** and **InvalidateRgn** functions do not actually generate WM_PAINT messages. Instead, Windows accumulates the changes made by these functions and its own changes while a window processes other messages in its application queue. Postponing the WM_PAINT message lets a window process all changes at once instead of updating bits and pieces in time-consuming individual steps.

A window can require Windows to send a WM_PAINT message by using the **UpdateWindow** function. The **UpdateWindow** function sends the message directly to the window, regardless of the number of other messages in the application queue. **UpdateWindow** is typically used when a window wants to update its client area immediately, such as just after the window is created.

Once a window receives a WM_PAINT message, it must call the **BeginPaint** function to retrieve the display context for the client area and to retrieve other information such as the update region and whether the background has been erased.

Windows automatically selects the update region as the clipping region of the display context. Since GDI discards (clips) drawing that extends outside the clipping region, only drawing that is in the update region is actually visible. For more information about the clipping region, see Section 2.8, "Clipping Functions."

The **BeginPaint** function empties the update region to prevent the same region from generating subsequent WM_PAINT messages.

After completing the painting operation, the window must call the **EndPaint** function to release the display context.

1.6.6 Update Region

An update region defines the part of the client area that is marked for painting on the next WM_PAINT message. The purpose of the update region is to save some applications the time it takes to paint the entire contents of the client area. If only the part that needs painting is added to the update region, only that part is painted. For example, if a word changes in the client area of a word-processing application, only the word needs to be painted, not the entire line of text. This saves the time it takes the application to draw the text, especially if there are many different sizes and typefaces.

The **InvalidateRect** and **InvalidateRgn** functions add a given rectangle or region to the update region. The rectangle or region must be given in client coordinates. The update region itself is defined in client coordinates. Windows adds its own rectangles and regions to a window's update region after operations such as moving, sizing, and scrolling the window.

The **ValidateRect** and **ValidateRgn** functions remove a given rectangle or region from the update region. These functions are typically used when the window has updated a specific part of the display in the update region before receiving the WM_PAINT message.

The **GetUpdateRect** and **GetUpdateRgn** functions retrieve the smallest rectangle that encloses the entire update region. These functions can be used to compute the current size of the update region to determine if painting is required.

1.6.7 Window Background

The window background is the color or pattern the client area is filled with before a window begins painting in the client area. Windows paints the background for a window or gives the window the opportunity to do so by sending a WM_ERASEBKGND message to the window when the application calls the **BeginPaint** function.

The background is important since if not erased, the client area will contain whatever was originally on the system display before the window was moved there. Windows erases the background by filling it with the background brush specified by the window's class.

Windows applications that use class or private display contexts should be careful about erasing the background. Windows assumes the background is to be computed by using the MM_TEXT mapping mode. If the display context has any other mapping mode, the area erased may not be within the visible part of the client area.

1.6.8 Brush Alignment

Brush alignment is particularly important on the system display where scrolling and moving are commonplace. A brush is a pattern of bits with a minimum size of 8-by-8 bits. GDI paints with a brush by repeating the pattern again and again within a given rectangle or region. If the region is moved by an arbitrary amount—for example, if the window is scrolled—and the brush is used again to filled empty areas around the original area, there is no guarantee that the original pattern and the new pattern will be aligned. For example, if the scroll moves the original filled area up one pixel, the intersection of the original area and any new painting will be out of alignment by one pixel, or bit. Depending on the pattern, this may have a undesirable visual effect.

To ensure that a brush is aligned after a window is moved, an application must take the following steps:

1. Call the **SelectObject** function to select a different brush.
2. Call the **SetBrushOrg** function to realign the current brush.
3. Call the **UnrealizeObject** function to realign the origin of the original brush when it is selected next.
4. Call the **SelectObject** function to select the original brush.

1.6.9 Painting Rectangular Areas

The **FillRect**, **FrameRect**, and **InvertRect** functions provide an easy way to carry out painting operations on rectangles in the client area.

The **FillRect** function fills a rectangle with the color and pattern of a given brush. This function fills all parts of the rectangle, including the edges or borders.

The **FrameRect** function uses a brush to draw a border around a rectangle. The border width and height is one unit.

The **InvertRect** function inverts the contents of the given rectangle. On monochrome displays, white pixels become black, and vice versa. On color displays, the results depend on the method used by the display to generate color. In either case, calling **InvertRect** twice with the same rectangle restores the display to its original colors.

1.6.10 Drawing Icons

The **DrawIcon** function draws an icon at a given location in the client area. An icon is a bitmap that a window uses as a symbol to represent an item or concept, such as an application or a warning.

An icon can be created by using the SDKPaint program, added to an application's resources by using the Resource Compiler, and loaded into memory by using the **LoadIcon** function. Applications can also call the **CreateIcon** function to create an icon and can modify a previously loaded or created icon at any time. An icon resource is in global memory and its handle is the handle to that memory. An application can free memory used to store an icon created by **CreateIcon** by calling **DeleteIcon**.

1.6.11 Drawing Formatted Text

The **DrawText** function formats and draws text within a given rectangle in the client area. This function provides simple text processing that most applications, other than word processors, can use to display text. **DrawText** output is similar to the output generated by a terminal, except it uses the selected font and can clip the text if it extends outside a given rectangle. **DrawText** provides many different formatting styles. Table 1.5 lists the styles that are available:

Table 1.5 Text Formatting Styles

Value	Description
DT_BOTTOM	Bottom-justified (single line only).
DT_CENTER	Centered.
DT_EXPANDTABS	Expands tab characters into spaces. Otherwise, tabs are treated as single characters. The number of spaces depends on the tab stop size specified by DT_TABSTOP. If DT_TABSTOP is not given, the default is eight spaces.
DT_EXTERNALLEADING	Includes the font external leading in line height. External leading is not included in the height of a line of text. (Leading is the space between lines of text.) If DT_EXTERNALLEADING is not given, there is no spacing between lines of text. Depending on the selected font, this means that characters in different lines may touch or overlap.
DT_LEFT	Left-justified. Default.
DT_NOCLIP	Draws text without clipping. All text will be drawn even if it extends outside the specified rectangle. The **DrawText** function is somewhat faster when DT_NOCLIP is used.
DT_RIGHT	Right-justified.
DT_SINGLELINE	Single line only. Carriage returns and linefeeds do not break the line. Default is multiple-line formatting.
DT_TABSTOP	Sets tab stops. The high-order byte of the *wFormat* parameter is the number of characters for each tab. If DT_TABSTOP is not given, the default tab size is eight spaces.
DT_TOP	Top-justified (single line only). Default.
DT_VCENTER	Vertically centered (single line only).
DT_WORDBREAK	Sets word breaks. Lines are automatically broken between words if a word would extend past the edge of the rectangle specified by the *lpRect* parameter. Carriage-return/linefeed sequence also causes a line break. Word-break characters are space, tab, carriage return, linefeed, and carriage-return/linefeed combinations. Applies to multiple-line formatting only.

The **DrawText** function uses the selected font, so applications can draw formatted text in other than the system font.

DrawText does not hyphenate, and although it can justify text to the left, right, or center, it cannot combine justification styles. In other words, it cannot justify both left and right.

DrawText recognizes a number of control characters and carries out special actions when it encounters them. Table 1.6 lists the control characters and the respective action:

Table 1.6 DrawText Control Characters

Character (ANSI value)	Action
Carriage return(13)	Interpreted as a line-break character. The text is immediately broken and started on the next line down in the rectangle.
Linefeed(10)	Interpreted as a line-break character. The text is immediately broken and started on the next line down in the rectangle.
	A carriage-return/linefeed character combination is interpreted as a single line-break character.
Space(32)	Interpreted as a word-break character if the DT_WORDBREAK style is given. If the text is too long to fit on the current line in the formatting rectangle, the line is broken at the closest word-break character to the end of the line.
Tab(9)	Expanded into a given number of spaces if the DT_EXPANDTABS style is given. The number of spaces depends on what tab-stop value is given with the DT_TABSTOP style. The default is eight.

1.6.12 Drawing Gray Text

An application can draw gray text by calling the **SetTextColor** function to set the current text color to the COLOR_GRAYTEXT, the solid gray system color used to draw disabled text. However, if the curent display driver does not support a solid gray color, this value is set to zero.

The **GrayString** function is a multiple-purpose function that gives applications another way to gray text or carry out other customized operations on text or bitmaps before drawing the result in a client area. To gray text, the function creates a memory bitmap, draws the string in the bitmap, and then grays the string by combining it with a gray brush. The **GrayString** function finally copies the gray text to the display. An application can intercept or modify each step of this process, however, to carry out custom effects, such as changing the gray brush to a patterned brush or drawing an icon instead of a string.

If **GrayString** is used to draw gray text only, **GrayString** uses the selected font of the given display context. **GrayString** sets text color to black. It creates a bitmap, and then uses the **TextOut** function to write a given string to the bitmap. It then uses the **PatBlt** function and a gray brush to gray the text, and uses the **BitBlt** function to copy the bitmap to the client area.

GrayString assumes that the display context for the client area has MM_TEXT mapping mode. Other mapping modes cause undesirable results.

GrayString lets an application modify this graying procedure in three ways: by defining an additional brush to be combined with the text before being displayed, by replacing the call to the **TextOut** function with a call to an application-supplied function, and by disabling the call to the **PatBlt** function.

The additional brush is defined as a parameter. This brush is combined with the text as the text is being copied to the client area by the **BitBlt** function. The additional brush is intended to be used to give the text a desired color, since the bitmap used to draw the text is a monochrome bitmap.

The application-supplied function is also defined as a parameter. If a non-NULL value is given for the function, **GrayString** automatically calls the application-supplied function instead of the **TextOut** function and passes it a handle to the display context for the memory bitmap as well as the long pointer and count passed to **GrayString**. The function can carry out any operation and interpret the long pointer and count in any way. For example, a negative count could be used to indicate that the long pointer points to an icon handle that signals the application-supplied function to draw the icon and let **GrayString** gray and display it. No matter what type of drawing the function carries out, **GrayString** assumes it is successful if the application-supplied function returns TRUE.

GrayString suppresses graying if it receives an *ncount* parameter equal to –1 and the application-supplied function returns FALSE. This is a way to combine custom patterns with the text without interference from the gray brush.

1.6.13 Nonclient-Area Painting

Windows sends a WM_NCPAINT message to the window whenever the nonclient area of the window, such as the title bar, menu bar, and window frame, needs painting. Processing this message is not recommended since a window that does so must be able to paint all the required parts of the nonclient area for the window. In other words, a window should pass this message on to the **DefWindowProc** function for default processing unless the Windows application is creating a custom nonclient area for a child window.

1.7 Dialog-Box Functions

Dialog-box functions create, alter, test, and destroy dialog boxes and controls within dialog boxes. A dialog box is a temporary window that Windows creates for special-purpose input, and then destroys immediately after use. An application typically uses a dialog box to prompt the user for additional information about a current command selection. The following list briefly describes each dialog function:

Function	Description
CheckDlgButton	Places/removes a check, or changes the state of the three-state button.
CheckRadioButton	Checks a specified button and removes checks from all others.
CreateDialog	Creates a modeless dialog box.
CreateDialogIndirect	Creates a modeless dialog box from a template.
CreateDialogIndirectParam	Creates a modeless dialog box from a template and passes data to it when it is created.
CreateDialogParam	Creates a modeless dialog box and passes data to it when it is created.
DefDlgProc	Provides default processing for any Windows messages that a dialog box with a private window class does not process.
DialogBox	Creates a modal dialog box.
DialogBoxIndirect	Creates a modal dialog box from a template.
DialogBoxIndirectParam	Creates a modal dialog box from a template and passes data to it when it is created.
DialogBoxParam	Creates a modal dialog box and passes data to it when it is created.
DlgDirList	Fills the list box with names of files matching a path.
DlgDirListComboBox	Fills a combo box with names of files matching a path.

Function	Description
DlgDirSelect	Copies the current selection from a list box to a string.
DlgDirSelectComboBox	Copies the current selection from a combo box to a string.
EndDialog	Frees resources and destroys windows associated with a modal dialog box.
GetDialogBaseUnits	Retrieves the base dialog units used by Windows when creating a dialog box.
GetDlgCtrlID	Returns the ID value of a control window.
GetDlgItem	Retrieves the handle of a dialog item from the given dialog box.
GetDlgItemInt	Translates the control text of an item into an integer value.
GetDlgItemText	Copies an item's control text into a string.
GetNextDlgGroupItem	Returns the window handle of the next item in a group.
GetNextDlgTabItem	Returns the window handle of the next or previous item.
IsDialogMessage	Determines whether a message is intended for the given dialog box.
IsDlgButtonChecked	Tests whether a button is checked.
MapDialogRect	Converts the dialog-box coordinates to client coordinates.
SendDlgItemMessage	Sends a message to an item within a dialog box.
SetDlgItemInt	Sets the caption or text of an item to a string that represents an integer.
SetDlgItemText	Sets the caption or text of an item to a string.

1.7.1 Uses for Dialog Boxes

For convenience and to keep from introducing device-dependent values into the application code, applications use dialog boxes instead of creating their own windows. This device independence is maintained by using logical coordinates in the dialog-box template. Dialog boxes are convenient to use because all aspects of the dialog box, except how to carry out its tasks, are predefined. Dialog boxes supply a window class and procedure, and create the window for the dialog box automatically. The application supplies a dialog function to carry out tasks and a dialog-box template that describes the dialog style and content.

Modeless Dialog Box

A modeless dialog box allows the user to supply information to the dialog box and return to the previous task without canceling or removing the dialog box. Modeless dialog boxes are typically used as a way to let the user continually supply information about the current task without having to select a command from a menu each time. For example, modeless dialog boxes are often used with a text-search command in word-processing applications. The dialog box remains displayed while the search is carried out. The user can then return to the dialog box and search for the same word again, or change the entry in the dialog box and search for a new word.

An application with a modeless dialog box processes messages for that box by using the **IsDialogMessage** function inside the main message loop. The dialog function of a modeless dialog box must send a message to the parent window when it has input for the parent window. It must also destroy the dialog box when it is no longer needed. A modeless dialog box can be destroyed by using the **DestroyWindow** function. An application must not call the **EndDialog** function to destroy a modeless dialog box.

Modal Dialog Box

A modal dialog box requires the user to respond to a request before the application continues. Typically, a modal dialog box is used when a chosen command needs additional information before it can proceed. The user should not be able to continue some other operation unless the command is canceled or additional information is provided.

A modal dialog box disables its parent window, and it creates its own message loop, temporarily taking control of the application queue from the main loop of the program. A modal dialog box is displayed when the application calls the **DialogBox** function.

By default, a modal dialog box cannot be moved by the user. An application can create a moveable dialog box by specifying the WS_CAPTION and, optionally, the WS_SYSMENU window styles.

The dialog box is displayed until the dialog function calls the **EndDialog** function, or until Windows is terminated. The parent window remains disabled unless the dialog box enables it. Note that enabling the parent window is not recommended since it defeats the purpose of the modal dialog box.

System-Modal Dialog Box

A system-modal dialog box is identical to a modal dialog box except that all windows, not just the parent window, are disabled. System-modal dialog boxes must be used with care since they effectively shut down the system until the user supplies the required information.

1.7.2 Creating a Dialog Box

A dialog box is created by using either the **CreateDialog** or **DialogBox** function. These functions load a dialog-box template from the application's executable file, and then create a pop-up window that matches the template's specifications. The dialog box belongs to the predefined dialog-box class unless another class is explicitly defined. The **DialogBox** function creates a modal dialog box; the **CreateDialog** function creates a modeless dialog box.

Use the WS_VISIBLE style for the dialog-box template if you want the dialog box to appear upon creation.

Dialog-Box Template

The dialog-box template is a description of the dialog box: its height and width, the controls it contains, its style, the type of border it uses, and so on. A template is an application's resource and must be added to the application's executable file by using the Resource Compiler.

Dialog boxes can be easily modified and are system independent, enabling an application developer to change the template without changing the source code.

The **CreateDialog** and **DialogBox** functions load the resource into memory when they create the dialog box, and then use the information in the dialog template to create the dialog box, position it, and create and position the controls for the dialog box.

The Resource Compiler takes a text description of the template and converts it to the required binary form. This binary form is added to the application's executable file.

Dialog-Box Measurements

Dialog box and control dimensions and coordinates are device independent. Since a dialog box may be displayed on system displays that have widely varying pixel resolutions, dialog-box dimensions are specified in system character widths and heights instead of pixels. Characters are guaranteed to give the best possible appearance for a given display. One unit in the *x* direction is equal to ¼ of the dialog base width unit. One unit in the *y* direction is equal to ⅛ of the dialog base height unit. The dialog base units are computed from the height and width of the system font; the **GetDialogBaseUnits** function returns the dialog base units for the current display. Applications can convert these measurements to pixels by using the **MapDialogRect** function.

Windows does not allow the height of a dialog box to exceed the height of a full-screen window. The width of a dialog box is not allowed to be greater than the width of the screen.

1.7.3 Return Values from a Dialog Box

The **DialogBox** function that creates a modal dialog box does not return until the dialog function has called the **EndDialog** function to signal the end of the dialog box. When control finally returns from the **DialogBox** function, the return value is equal to the value specified in the **EndDialog** function. This means a modal dialog box can return a value through the **EndDialog** function.

Modeless dialog boxes cannot return values in this way since they do not use the **EndDialog** function to terminate execution and do not return control in the same way a modal dialog box does. Instead, modeless dialog boxes return values to their parent windows by using the **SendMessage** function to send a notification message to the parent window. Although Windows does not explicitly define the content of a notification message, most applications use a WM_COMMAND message with an integer value that identifies the dialog box in the *wParam* parameter and the return value in the *lParam* parameter. Modal dialog boxes may also use this technique to return values to their parent windows before terminating.

1.7.4 Controls in a Dialog Box

A dialog box can contain any number and any type of controls. A control is a child window that belongs to a predefined or application-defined window class and that gives the user a method of supplying input to the application. Examples of controls are push buttons and edit controls. Most dialog boxes contain one or more controls of the predefined class. The number of controls, the order in which they should be created, and the location of each in the dialog box are defined by the control statements given in the dialog-box template.

Control Identifiers

Every control in a dialog box needs a unique control identifier, or ID, to distinguish it from other controls. Since all controls send information to the dialog function through WM_COMMAND messages, the control identifiers are essential for the dialog box to determine which control sent a given message.

All identifiers for all controls in the dialog box must be unique. If a dialog box has a menu bar, there must be no conflict between menu-item identifiers and control identifiers. Since Windows sends menu input to a dialog function as WM_COMMAND messages, conflicts with menu and control identifiers can cause errors. Menus in dialog boxes are not recommended.

The dialog function usually identifies the dialog-box controls by using their control identifier. Occasionally the dialog function requires the window handle that was given to the control when it was created. The dialog function can retrieve this window handle by using the **GetDlgItem** function.

General Control Styles

The WS_TABSTOP style specifies that the user can move the input focus to the given control by pressing the TAB or SHIFT+TAB keys. Typically, every control in the dialog box has this style, so the user can move the input focus from one control to the other. If two or more controls are in the dialog box, the TAB key moves the input focus to the controls in the order in which they have been created. The SHIFT+TAB keys move the input focus in reverse order. For modal dialog boxes, the TAB and SHIFT+TAB keys are automatically enabled for moving the input focus. For modeless dialog boxes, the **IsDialogMessage** function must be used to filter messages for the dialog box and to process these key strokes. Otherwise, the keys have no special meaning and the WS_TABSTOP style is ignored.

The WS_GROUP style specifies that the user can move the input focus to the given control by using a DIRECTION key. Typically, the first and last controls in a group of consecutive controls in the dialog box have this style, so the user can move the input focus from one control to the other. The DOWN and RIGHT keys move the input focus to controls in the order in which they have been created. The UP and LEFT keys move the input focus in reverse order. For modal dialog boxes, the DIRECTION keys are automatically enabled for moving the input focus. For modeless dialog boxes, the **IsDialogMessage** function must be used to filter messages for the dialog box and to process these key strokes. Otherwise, the keys have no special meaning and the WS_GROUP style is ignored.

Buttons

Button controls are the principal interface of a dialog box. Almost all dialog boxes have at least one push-button control and most have one default push button and one or more other push buttons. Many dialog boxes have collections of radio buttons enclosed in group boxes, or lists of check boxes.

Most modal or modeless dialog boxes that use the special keyboard interface have a default push button whose control identifier is set to 1 so that the action the dialog function takes when the button is clicked is identical to the action taken when the ENTER key is pressed. There can be only one button with the default style; however, an application can assign the default style to any button at any time. These dialog boxes may also set the control identifier of another push button to 2 so that the action of the ESCAPE key is duplicated by clicking that button.

When a dialog box first starts, the dialog function can set the initial state of the button controls by using the **CheckDlgButton** function, which sets or clears the button state. This function is most useful when used to set the state of radio buttons or check boxes. If the dialog box contains a group of radio buttons in which only one button should be set at any given time, the dialog function can use the **CheckRadioButton** function to set the button and automatically clear any other radio button.

Before a dialog box terminates, the dialog function can check the state of each button control by using the **IsDlgButtonChecked** function, which returns the current state of the button. A dialog box typically saves this information to initialize the buttons the next time the dialog box is created.

Edit Controls

Many dialog boxes have edit controls that let the user supply text as input. Most dialog functions initialize an edit control when the dialog box first starts. For example, the function may place a proposed filename in the control that the user can adapt or modify. The dialog function can set the text in an edit control by using the **SetDlgItemText** function, which copies text in a given buffer to the edit control. When the edit control receives the input focus, the complete text will automatically be selected for editing.

Since edit controls do not automatically return their text to the dialog box, the dialog function must retrieve the text before terminating. It can retrieve the text by using the **GetDlgItemText** function, which copies the edit-control text to a buffer. The dialog function typically saves this text to initialize the edit control later, or passes it on to the parent window for processing.

Some dialog boxes use edit controls that let the user enter numbers. The dialog function can retrieve a number from an edit control by using the **GetDlgItemInt** function, which retrieves the text of the control and converts the text to a decimal value. The user enters the number in decimal digits. It can be either signed or unsigned. The dialog function can display an integer by using the **SetDlgItemInt** function. It converts a signed or unsigned integer to a string of decimal digits.

List Boxes and Directory Listings

Some dialog boxes display lists, such as filenames, from which the user can select one or more names. Dialog boxes that display a list typically use list-box controls. Dialog boxes that display a list of filenames typically use a list-box control and the **DlgDirList** and **DlgDirSelect** functions. The **DlgDirList** function automatically fills a list box with the filenames in the current directory. The **DlgDirSelect** function retrieves the selected filename from the list box. Together they provide a convenient way for a dialog box to display a directory listing, and let the user select a file without having to type in the name of the directory and file.

Combo Boxes

Another method for providing a list of items to a user is by means of a combo box. A combo box consists of either a static text field or edit field combined with a list box. The list box can be displayed at all times or pulled down by the user. If the combo box contains a static text field, the text field always displays the current selection (if any) in the list-box portion of the combo box. If it uses an edit field, the user can type in the desired selection; the list box highlights the first item (if any) which matches what the user has entered in the edit field. The user can then select the item highlighted in the list box to complete the choice.

Owner-Draw Dialog-Box Controls

List boxes, combo boxes, and buttons can be designated as owner-draw controls by creating them with the appropriate style:

Style	Meaning
LBS_OWNERDRAWFIXED	Creates an owner-draw list box with items that have the same, fixed height.
LBS_OWNERDRAWVARIABLE	Creates an owner-draw list box with items that have different heights.
CBS_OWNERDRAWFIXED	Creates an owner-draw combo box with items that have the same, fixed height.
CBS_OWNERDRAWVARIABLE	Creates an owner-draw combo box with items that have different heights.
BS_OWNERDRAW	Creates an owner-draw button.

When a control has the owner-draw style, Windows handles the user's interaction with the control as usual, such as detecting when a user has clicked a button and notifying the button's owner of the event. However, because it is an owner-draw control, the owner of the control is completely responsible for the visual appearance of the control.

When Windows first creates a dialog box containing owner-draw controls, it sends the owner a WM_MEASUREITEM message for each owner-draw control. The *lParam* parameter of this message contains a pointer to a **MEASURE-ITEMSTRUCT** data structure. When the owner receives the message for a control, the owner fills in the appropriate fields of the structure and returns. This informs Windows of the dimensions of the control or of its items so that Windows can appropriately detect the user's interaction with the control. If a list box or combo box is created with the LBS_OWNERDRAWVARIABLE or CBS_OWNERDRAWVARIABLE style, this message is sent to the owner for each item in the control, since each item can differ in height. Otherwise, this message is sent once for the entire owner-draw control.

Whenever an owner-draw control needs to be redrawn, Windows sends the WM_DRAWITEM message to the owner of the control. The *lParam* parameter of this message contains a pointer to a **DRAWITEMSTRUCT** data structure that contains information about the drawing required for the control. Similarly, if an item is deleted from a list box or combo box, Windows sends the WM_DELETEITEM message containing a pointer to a **DELETEITEM-STRUCT** data structure that describes the deleted item.

Messages for Dialog-Box Controls

Many controls recognize predefined messages that, when sent to the control, cause it to carry out some action. A dialog function can send a message to a control by supplying the control identifier and using the **SendDlgItemMessage** function, which is identical to the **SendMessage** function except that it uses a control identifier instead of a window handle to identify the control that is to receive the message.

1.7.5 Dialog-Box Keyboard Interface

Windows provides a special keyboard interface for modal dialog boxes and modeless dialog boxes that use the **IsDialogMessage** function to filter messages. This keyboard interface carries out special processing for several keys and generates messages that correspond to certain buttons in the dialog box or changes the input focus from one control to another. Table 1.7 lists the keys used in this interface and the respective action:

Table 1.7 Dialog-Box Keyboard Interface

Key	Action
DOWN	Moves the input focus to the next control that has the WS_GROUP style.
ENTER	Sends a WM_COMMAND message to the dialog function. The *wParam* parameter is set to 1 or the default button.
ESCAPE	Sends a WM_COMMAND message to the dialog function. The *wParam* parameter is set to 2.
LEFT	Same as UP.
RIGHT	Same as DOWN.
SHIFT+TAB	Moves the input focus to the previous control that has the WS_TABSTOP style.

Table 1.7 Dialog-Box Keyboard Interface *(continued)*

Key	Action
TAB	Moves the input focus to the next control that has the WS_TAB-STOP style.
UP	Moves the input focus to the previous control that has the WS_GROUP style.

The TAB and DIRECTION keys have no effect if the controls in the dialog box do not have the WS_TABSTOP or WS_GROUP style. The keys have no effect in a modeless dialog box if the **IsDialogMessage** function is not used to filter messages for the dialog box.

NOTE For applications that use accelerators and have modeless dialog boxes, the **IsDialogMessage** function must be called before the **TranslateAccelerator** function. Otherwise, the keyboard interface for the dialog box may not be processed correctly.

Applications that have modeless dialog boxes and want those boxes to have the special keyboard interface must filter all messages retrieved from the application queue through the **IsDialogMessage** function before carrying out any other processing. This means that the application must pass the message to the function immediately after retrieving the message by using the **GetMessage** or **PeekMessage** function. Most applications that have modeless dialog boxes incorporate the **IsDialogMessage** function as part of the main message loop in the WinMain function. The **IsDialogMessage** function automatically processes any messages for the dialog box. This means that if the function returns a nonzero value, the message does not require additional processing and must not be passed to the **TranslateMessage** or **DispatchMessage** function.

The **IsDialogMessage** function also processes the ALT+*mnemonic* sequence.

Scrolling in Dialog Boxes

In modal dialog boxes, the DIRECTION keys have specific functions that depend on the controls in the box. For example, the keys move the input focus from control to control in group boxes, move the cursor in edit controls, and scroll the contents of list boxes. The DIRECTION keys cannot be used to scroll the contents of any dialog box that has its own scroll bars. If a dialog box has scroll bars, the application must provide an appropriate keyboard interface for the scroll bars. Note that the mouse interface for scrolling is available if the system has a mouse.

1.8 Scrolling Functions

Scrolling functions control the scrolling of a window's contents and control the window's scroll bars. Scrolling is the movement of data in and out of the client area at the request of the user. It is a way for the user to see a document or graphic in parts if Windows cannot fit the entire document or graphic inside the client area. A scroll bar allows the user to control scrolling. The following list briefly describes each scrolling function:

Function	Description
GetScrollPos	Retrieves the current position of the scroll-bar thumb.
GetScrollRange	Copies the minimum and maximum scroll-bar positions for given scroll-bar positions for a specified scroll.
ScrollDC	Scrolls a rectangle of bits horizontally and vertically.
ScrollWindow	Moves the contents of the client area.
SetScrollPos	Sets the scroll-bar thumb.
SetScrollRange	Sets the minimum and maximum scroll-bar positions.
ShowScrollBar	Displays or hides a scroll bar and its controls.

1.8.1 Standard Scroll Bars and Scroll-Bar Controls

A standard scroll bar is a part of the nonclient area of a window. It is created with the window and displayed when the window is displayed. The sole purpose of a standard scroll bar is to let users generate scrolling requests for the window's client area. A window has standard scroll bars if it is created with the WS_VSCROLL or WS_HSCROLL style. A standard scroll bar is either vertical or horizontal. A vertical bar always appears at the right of the client area; a horizontal bar always appears at the bottom. A standard scroll bar always has the standard scroll-bar height and width as defined by the SM_CXVSCROLL and SM_CYHSCROLL system metric values. (For more information, see the **GetSystemMetrics** function in Chapter 4, "Functions Directory.")

A scroll-bar control is a control window that looks and acts like a standard scroll bar. But unlike a standard scroll bar, a scroll-bar control is not part of any window. As a separate window, a scroll-bar control can receive the input focus, and indicates this by displaying a flashing caret in the thumb. When a scroll-bar control has the input focus, the user can use the keyboard to direct the scrolling. Unlike standard scroll bars, a scroll-bar control provides a built-in keyboard interface. Scroll-bar controls also can be used for other purposes. For example, a scroll-bar control can be used to select values from a range of values, such as a color from a rainbow of colors.

1.8.2 Scroll-Bar Thumb

The scroll-bar thumb is the small rectangle in a scroll bar. It shows the approximate location within the current document or file of the data currently displayed in the client area. For example, the thumb is in the middle of the scroll bar when page three of a five-page document is in the client area.

The **SetScrollPos** function sets the thumb position in a scroll bar. Since Windows does not automatically update the thumb position when an application scrolls, **SetScrollPos** must be used to update the thumb position. The **GetScrollPos** function retrieves the current position.

A thumb position is an integer. The position is relative to the left or upper end of the scroll bar, depending on whether the scroll bar is horizontal or vertical. The position must be within the scroll-bar range, which is defined by minimum and maximum values. The positions are distributed equally along the scroll bar. For example, if the range is 0 to 100, there are 100 positions along the scroll bar, each equally spaced so that position 50 is in the middle of the scroll bar. The initial range depends on the scroll bar. Standard scroll bars have an initial range of 0 to 100; scroll-bar controls have an empty range (both minimum and maximum values are zero) if no explicit range is given when the control is created. The **SetScrollRange** function sets new minimum and maximum values so that applications can change the range at any time. The **GetScrollRange** function retrieves the current minimum and maximum values. The minimum and maximum values can be any integers. For example, a spreadsheet program with 255 rows can set the vertical scroll range to 1 to 255.

If **SetScrollPos** specifies a position value that is less than the minimum or more than the maximum, the minimum or maximum value is used instead. **SetScrollPos** moves the thumb along the thumb positions.

1.8.3 Scrolling Requests

A user makes a scrolling request by clicking in a scroll bar. Windows sends the request to the given window in the form of WM_HSCROLL and WM_VSCROLL messages. The *lParam* parameter contains a position value and the handle of the scroll-bar control that generated the message (*lParam* is zero if a standard scroll bar generated the message). The *wParam* parameter specifies the type of scroll, such as scroll up one line, scroll down a page, or scroll to the bottom. The type of scroll is determined by which area of the scroll bar the user clicks.

The user can also make a scrolling request by using the scroll-bar thumb, the small rectangle inside the scroll bar. The user moves the thumb by moving the mouse while holding the left mouse button down when the cursor is in the thumb. The scroll bar sends SB_THUMBTRACK and SB_THUMBPOSITION flags with a WM_HSCROLL or WM_VSCROLL message to an application as the user moves the thumb. Each message specifies the current position of the thumb.

1.8.4 Processing Scroll Messages

A window that permits scrolling needs a standard scroll bar or a scroll-bar control to let the user generate scrolling requests, and a window function to process the WM_HSCROLL and WM_VSCROLL messages that represent the scrolling requests. Although the result of a scrolling request is entirely up to the window, a window typically carries out a scroll by moving in some direction from the current location or to a known beginning or end, and by displaying the data at the new location. For example, a word-processing application can scroll to the next line, the next page, or to the end of the document.

1.8.5 Scrolling the Client Area

The simplest way to scroll is to erase the current contents of the client area, and then paint the new information. This is the method an application is likely to use with SB_PAGEUP, SB_PAGEDOWN, SB_TOP, and SB_END requests where completely new contents are required.

For some requests, such as SB_LINEUP and SB_LINEDOWN, not all the contents need to be erased, since some will still be visible after the scroll. The **ScrollWindow** function preserves a portion of the client area's contents, moves the preserved portion the specified amount, and prepares the rest of the client area for painting new information. **ScrollWindow** uses the **BitBlt** function to move a specific part of the client area to a new location within the client area. Any part of the client area that is uncovered (not in the part to be preserved) is invalidated and will be erased and painted over at the next WM_PAINT message.

ScrollWindow also lets an application clip a part of the client area from the scroll. This is to keep items that have fixed positions in the client area, such as child windows, from moving. This action automatically invalidates the part of the client area that is to receive the new information so that the application does not have to compute its own clipping regions.

1.8.6 Hiding a Standard Scroll Bar

For standard scroll bars, if the minimum and maximum values are equal, the scroll bar is considered disabled and is hidden. This is the way to temporarily hide a scroll bar when it is not needed for the current contents of the client area.

The **SetScrollRange** function hides and disables a standard scroll bar when it sets the minimum and maximum values to equal values. No scrolling requests can be made through the scroll bar when it is hidden. **SetScrollRange** enables the scroll bar and shows it again when it sets the minimum and maximum values to unequal values. The **ShowScrollBar** function can also be used to hide or show a scroll bar. It does not affect the scroll bar's range or thumb position.

1.9 Menu Functions

Menu functions create, modify, and destroy menus. A menu is an input tool in a Windows application that offers users one or more choices, which they can select with the mouse or keyboard. An item in a menu bar can display a pop-up menu, and any item in a pop-up menu can display another pop-up menu. In addition, a pop-up menu can appear anywhere on the screen. The following list briefly describes each menu function:

Function	Description
AppendMenu	Appends a menu item to a menu.
CheckMenuItem	Places or removes checkmarks next to pop-up menu items.
CreateMenu	Creates an empty menu.
CreatePopupMenu	Creates an empty pop-up menu.
DeleteMenu	Removes a menu item and destroys any associated pop-up menus.
DestroyMenu	Destroys the specified menu.
DrawMenuBar	Redraws a menu bar.
EnableMenuItem	Enables, disables, or grays a menu item.
GetMenu	Retrieves a handle to the menu of a specified window.
GetMenuCheckMarkDimensions	Retrieves the dimensions of the default menu checkmark bitmap.
GetMenuItemCount	Returns the count of items in a menu.
GetMenuItemID	Returns the item's identification.
GetMenuState	Obtains the status of a menu item.
GetMenuString	Copies a menu label into a string.
GetSubMenu	Retrieves the menu handle of a pop-up menu.

Function	Description
GetSystemMenu	Accesses the System menu for copying and modification.
HiliteMenuItem	Highlights or removes the highlighting from a top-level (menu-bar) menu item.
InsertMenu	Inserts a menu item in a menu.
LoadMenuIndirect	Loads a menu resource.
ModifyMenu	Changes a menu item.
RemoveMenu	Removes an item from a menu but does not destroy it.
SetMenu	Specifies a new menu for a window.
SetMenuItemBitmaps	Associates bitmaps with a menu item for display when an item is and is not checked.
TrackPopupMenu	Displays a pop-up menu at a specified screen location and tracks user interaction with the menu.

1.10 Information Functions

Information functions obtain information about the number and position of windows on the screen. The following list briefly describes each information function:

Function	Description
AnyPopup	Indicates whether any pop-up window exists.
ChildWindowFromPoint	Determines which child window contains a specific point.
EnumChildWindows	Enumerates the child windows that belong to a specific parent window.
EnumTaskWindows	Enumerates all windows associated with a given task.
EnumWindows	Enumerates windows on the display.
FindWindow	Returns the handle of a window with the given class and caption.

Function	Description
GetNextWindow	Returns a handle to the next or previous window.
GetParent	Retrieves the handle of the specified window's parent window.
GetTopWindow	Returns a handle to the top-level child window.
GetWindow	Returns a handle from the window manager's list.
GetWindowTask	Returns the handle of a task associated with a window.
IsChild	Determines whether a window is the descendent of a specified window.
IsWindow	Determines whether a window is a valid, existing window.
SetParent	Changes the parent window of a child window.
WindowFromPoint	Identifies the window containing a specified point.

1.11 System Functions

System functions return information about the system metrics, color, and time. The following list briefly describes each system function:

Function	Description
GetCurrentTime	Returns the time elapsed since the system was booted.
GetSysColor	Retrieves the system color.
GetSystemMetrics	Retrieves information about the system metrics.
SetSysColors	Changes one or more system colors.

1.12 Clipboard Functions

Clipboard functions carry out data interchange between Windows applications. The clipboard is the place for this interchange; it provides a place from which applications can pass data handles to other applications. The following list briefly describes each clipboard function:

Function	Description
ChangeClipboardChain	Removes a window from the chain of clipboard viewers.
CloseClipboard	Closes the clipboard.
EmptyClipboard	Empties the clipboard and reassigns clipboard ownership.
EnumClipboardFormats	Enumerates the available clipboard formats.
GetClipboardData	Retrieves data from the clipboard.
GetClipboardFormatName	Retrieves the clipboard format.
GetClipboardOwner	Retrieves the window handle associated with the current clipboard owner.
GetClipboardViewer	Retrieves the handle of the first window in the clipboard viewer chain.
GetPriorityClipboardFormat	Retrieves data from the clipboard in the first format in a prioritized format list.
IsClipboardFormatAvailable	Returns TRUE if the data in the given format is available.
OpenClipboard	Opens the clipboard.
RegisterClipboardFormat	Registers a new clipboard format.
SetClipboardData	Copies a handle for data.
SetClipboardViewer	Adds a handle to the clipboard viewer chain.

1.13 Error Functions

Error functions display errors and prompt the user for a response. The following list briefly describes each error function:

Function	Description
FlashWindow	Flashes the window by inverting its active/inactive state.
MessageBeep	Generates a beep on the system speaker.
MessageBox	Creates a window with the given text and caption.

1.14 Caret Functions

Caret functions affect the Windows caret, which is a flashing line, block, or bitmap that marks a location in a window's client area. The caret is especially useful in word-processing applications to mark a location in text for keyboard editing. These functions create, destroy, display, hide, and alter the blink time of the caret. The following list briefly describes each caret function:

Function	Description
CreateCaret	Creates a caret.
DestroyCaret	Destroys the current caret.
GetCaretBlinkTime	Returns the caret flash rate.
GetCaretPos	Returns the current caret position.
HideCaret	Removes a caret from a given window.
SetCaretBlinkTime	Establishes the caret flash rate.
SetCaretPos	Moves a caret to the specified position.
ShowCaret	Displays the newly created caret or redisplays a hidden caret.

1.14.1 Creating and Displaying a Caret

Windows forms a caret by inverting the pixel color within the rectangle given by the caret's position and its width and height. Windows flashes the caret by alternately inverting the display, and then restoring it to its previous appearance. The caret blink time (in milliseconds) defines the elapsed time between inverting and restoring the display. A complete flash (on-off-on) takes twice the blink time.

The **CreateCaret** function creates the caret shape and assigns ownership of the caret to the given window. The caret can be solid or gray, or, for bitmap carets, any desired pattern. The caret can have any shape, but typical shapes are a line, a solid block, a gray block, and a pattern, as shown in Figure 1.1:

Underline

Vertical line|

Solid block

Gray bloc

Bitmap

Figure 1.1 Caret Shapes

Windows displays a solid caret by inverting everything in the rectangle defined by the caret's width and height. For a gray caret, Windows inverts every other pixel. For a pattern, Windows inverts only the white bits of the bitmap that defines the pattern. The width and height of a caret are given in logical units, which means they are subject to the window's mapping mode.

1.14.2 Sharing the Caret

There is only one caret, so only one caret shape can be active at a time. Applications must cooperatively share the caret to prevent undesired effects. Windows does not inform an application when a caret is created or destroyed, so to be cooperative a window should create, move, show, and hide a caret only when it has the input focus or is active. A window should destroy the caret before losing the input focus or becoming inactive.

Bitmaps for the caret can be created by using the **CreateBitmap** function, or loaded from the application's resources by using the **LoadBitmap** function. Bitmaps loaded from resources can be created by using the SDKPaint program and added to an application's resources by using the Resource Compiler. (For more information about the Resource Compiler, see *Tools*.)

1.15 Cursor Functions

Cursor functions set, move, show, hide, and confine the cursor. The cursor is a bitmap, displayed on the display screen, that shows a current location. The following list briefly describes each cursor function:

Function	Description
ClipCursor	Restricts the cursor to a given rectangle.
CreateCursor	Creates a cursor from two bit masks.
DestroyCursor	Destroys a cursor created by the **CreateCursor** function.
GetCursorPos	Stores the cursor position (in screen coordinates).
LoadCursor	Loads a cursor from the resource file.
SetCursor	Sets the cursor shape.
SetCursorPos	Sets the position of the cursor.
ShowCursor	Increases or decreases the cursor display count.

1.15.1 Pointing Devices and the Cursor

When a system has a mouse (or any other type of pointing device), the cursor shows the current location of the mouse. Windows automatically displays and moves the cursor when the mouse is moved. If a system does not have a mouse, Windows does not automatically display or move the cursor. Applications can use the cursor functions to display or move the cursor when a system does not have a mouse.

1.15.2 Displaying and Hiding the Cursor

In a system without a mouse, Windows does not display or move the cursor unless the user chooses certain system commands, such as commands for sizing and moving. This means that after a call to **SetCursor**, the cursor remains on the screen until a subsequent call to **SetCursor** with a NULL parameter removes the cursor, or until a system command is carried out. Applications that wish to use the cursor without a mouse usually simulate mouse input by using keyboard keys, such as the DIRECTION keys, and display and move the cursor by using the cursor functions.

The **ShowCursor** function shows or hides the cursor. It is used to temporarily hide the cursor, and then restore it without changing the current cursor shape. This function actually sets an internal counter that determines whether the cursor should be drawn. Hiding and showing are accumulative, so hiding the cursor five times requires that it be shown five times before the cursor will be drawn.

1.15.3 Positioning the Cursor

The **SetCursorPos** and **GetCursorPos** functions set and retrieve the current screen coordinates of the cursor. Although the cursor can be set at a location other than the current mouse location, if the system has a mouse, the next mouse movement will redraw the cursor at the mouse location. The **SetCursorPos** and **GetCursorPos** functions are most often used in applications that use the keyboard and specified key strokes to move the cursor. Notice that screen coordinates are not affected by the mapping mode in a window's client area.

1.15.4 The Cursor Hotspot and Confining the Cursor

A cursor has a hotspot. When Windows draws the cursor, it always places the hotspot over the point on the display screen that represents the current position of the mouse or keyboard DIRECTION key. For example, the hotspot on the pointer is the point at the tip of the arrow.

The **ClipCursor** function confines the cursor to a given rectangle on the display screen. The cursor can move to the edge of the rectangle but cannot move out of it. **ClipCursor** is typically used to restrict the cursor to a given window such as a dialog box that contains a warning about a serious error. The rectangle is always given in screen coordinates and does not have to be within the window of the currently running application.

1.15.5 Creating a Custom Cursor

The **SetCursor** function sets the cursor shape and draws the cursor. When a system has a mouse, Windows automatically changes the shape of the cursor when it crosses a window border or enters a different part of a window, such as a title or menu bar. It uses standard cursor shapes for the different parts of the screen, such as a pointer in a title bar. The **SetCursor** function lets an application delete the standard cursor and draw its own custom cursor. The cursor keeps its new shape until the mouse moves or a system command is carried out.

1.16 Hook Functions

Hook functions manage system hooks, which are shared resources that install a specific type of filter function. A filter function is an application-supplied callback function, specified by the **SetWindowsHook** function, that processes events before they reach any application's message loop. Windows sends messages generated by a specific type of event to filter functions installed by the same type of hook. The following list briefly describes each hook function:

Function	Description
CallMsgFilter	Passes a message and other data to the current message-filter function.
DefHookProc	Calls the next filter function in a filter-function chain.

Function	Description
SetWindowsHook	Installs a system and/or application filter function.
UnhookWindowsHook	Removes a Windows filter function from a filter-function chain.

1.16.1 Filter-Function Chain

A filter-function chain is a series of connected filter functions for a particular system hook. For example, all keyboard filter functions are installed by WH_KEYBOARD and all journaling-record filter functions are installed by WH_JOURNALRECORD. Applications pass these filter functions to the system hooks with calls to the **SetWindowsHook** function. Each call adds a new filter function to the beginning of the chain. Whenever an application passes a filter function to a system hook, it must reserve space for the address of the next filter function in the chain. **SetWindowsHook** returns this address.

Once each filter function completes its task, it must call the **DefHookProc** function. **DefHookProc** uses the address stored in the location reserved by the application to access the next filter function in the chain.

To remove a filter function from a filter chain, an application must call the **UnhookWindowsHook** function with the type of hook and a pointer to the function.

There are five types of standard window hooks and two types of debugging hooks. Table 1.8 lists each type and describes its purpose:

Table 1.8 System Hooks

Type	Purpose
WH_CALLWNDPROC	Installs a window function filter.
WH_GETMESSAGE	Installs a message filter (on debugging versions only).
WH_JOURNALPLAYBACK	Installs a journaling playback filter.
WH_JOURNALRECORD	Installs a journaling record filter.
WH_KEYBOARD	Installs a keyboard filter.
WH_MSGFILTER	Installs a message filter.
WH_SYSMSGFILTER	Installs a system-wide message filter.

NOTE The WH_CALLWNDPROC and WH_GETMESSAGE hooks will affect system performance. They are supplied for debugging purposes only.

1.16.2 Installing a Filter Function

To install a filter function, an application must do the following:

1. Export the function in its module definition file.
2. Obtain the function's address by using the **MakeProcInstance** function.
3. Call the **SetWindowsHook** function, specifying the type of hook function (see Table 1.8, "System Hooks") and the address of the function (returned by **MakeProcInstance**).
4. Store the return value from **SetWindowsHook** in a reserved location. This value is the address of the previous filter function.

NOTE Filter functions and the return value from **SetWindowsHook** must reside in fixed library code and data. This allows these hooks to operate in a large-frame EMS environment.

1.17 Property Functions

Property functions create and access a window's property list. A property list is a storage area that contains handles for data that the application wishes to associate with a window. The following list briefly describes each property function:

Function	Description
EnumProps	Passes the properties of a window to an enumeration function.
GetProp	Retrieves a handle associated with a string from the window property list.
RemoveProp	Removes a string from the property list.
SetProp	Copies a string and a data handle to a window's property list.

Using Property Lists

Once a data handle is in a window's property list, any application can access the handle if it can also access the window. This makes the property list a convenient way to make data (for example, alternate captions or menus for the window) available to the application when it wishes to modify the window.

Every window has its own property list. When the window is created, the list is empty. The **SetProp** function adds entries to the list. Each entry contains a unique ANSI string and a data handle. The ANSI string identifies the handle; the handle identifies the data associated with the window, as illustrated in Figure 1.2:

ANSI String	Handle
"binary data"	*hMemory*
"icon"	*hicon*
"screen text"	*hText*
⋮	⋮

Figure 1.2 Property List

The data handle can identify any object or memory block that the application wishes to associate with the window. The **GetProp** function retrieves the data handle of an entry from the list without removing the entry. The handle can then be used to retrieve or use the data. The **RemoveProp** function removes an entry from the list when it is no longer needed.

Although the purpose of the property list is to associate data with a window for use by the application that owns the window, the handles in a property list are actually accessible to any application that has access to the window. This means an application can retrieve and use a data handle from the property list of a window created by another application. But using another application's data handles must be done with care. Only shared, global memory objects, such as GDI drawing objects, can be used by other applications. If a property list contains local or global memory handles or resource handles, only the application that has created the window may use them. Global memory handles can be shared with other applications by using the Windows clipboard. (For more information, see Section 1.12, "Clipboard Functions.") Local memory handles cannot be shared.

The contents of a property list can be enumerated by using the **EnumProps** function. The function passes the string and data handle of each entry in the list to an application-supplied function. The application-supplied function can carry out any task.

The data handles in a property list always belong to the application that created them. The property list itself, like other window-related data, belongs to Windows. A window's property list is actually allocated in the the USER heap, the local heap of the USER library. Although there is no defined limit to the number of entries in a property list, the actual number of entries depends on how much room is available in the USER heap. This depends on how many windows, window classes, and other window-related objects have been created.

The application creates the entries in a property list. Before a window is destroyed or the application that owns the window terminates, all entries in the property list must be removed by using the **RemoveProp** function. Failure to remove the entries leaves the property list in the USER heap and makes the space it occupies unusable for subsequent applications. This can ultimately cause an overflow of the USER heap. Entries in the property list can be removed at any time by using the **RemoveProp** function. If there are entries in the property list when the WM_DESTROY message is received for the window, the entries must be removed at that time. To ensure that all entries are removed, use the **EnumProps** function to enumerate all entries in the property list. An application should remove only those properties that it added to the property list. Windows adds properties for its own use and disposes of them automatically. An application must not remove properties which Windows has added to the list.

1.18 Rectangle Functions

Rectangle functions alter and obtain information about rectangles in a window's client area. In Windows, a rectangle is defined by a **RECT** data structure. The structure contains two points: the upper-left and lower-right corners of the rectangle. The sides of a rectangle extend from these two points and are parallel to the *x*- and *y*-axes. The following list briefly describes each rectangle function:

Function	Description
CopyRect	Makes a copy of an existing rectangle.
EqualRect	Determines whether two rectangles are equal.
InflateRect	Expands or shrinks the specified rectangle.
IntersectRect	Finds the intersection of two rectangles.
OffsetRect	Moves a given rectangle.
PtInRect	Indicates whether a specified point lies within a given rectangle.
SetRectEmpty	Sets a rectangle to an empty rectangle.
UnionRect	Stores the union of two rectangles.

1.18.1 Using Rectangles in a Windows Application

Rectangles are used to specify rectangular areas on the display or in a window, such as the cursor clipping area, the client repaint area, a formatting area for formatted text, and the scroll area. Rectangles are also used to fill, frame, or invert an area in the client area with a given brush, and to retrieve the coordinates of a window or a window's client area.

Since rectangles are used for many different purposes, the rectangle functions do not use an explicit unit of measure. Instead, all rectangle coordinates and dimensions are given in signed, logical values. The actual units are determined by the function in which the rectangle is used.

1.18.2 Rectangle Coordinates

Coordinate values for a rectangle can be within the range –32,768 to 32,767. Widths and heights, which must be positive, are within the range 0 to 32,767. This means that a rectangle whose left and right sides or whose top and bottom are further apart than 32,768 units is not valid. Figure 1.3 shows a rectangle whose upper-left corner is left of the origin, but whose width is less than 32,767:

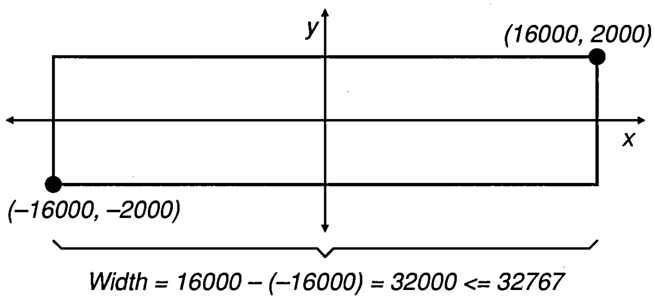


Figure 1.3 Rectangle Limits

1.18.3 Creating and Manipulating Rectangles

The **SetRect** function creates a rectangle, the **CopyRect** function makes a copy of a given rectangle, and the **SetRectEmpty** function creates an empty rectangle. An empty rectangle is any rectangle that has zero width, zero height, or both.

The **InflateRect** function increases or decreases the width and height of a rectangle. It adds or removes width from both ends of the rectangle, or adds or removes height from both the top and bottom of the rectangle.

The **OffsetRect** function moves the rectangle by a given amount. It moves the corners of the rectangle by adding the given *x* and *y* amounts to the corner coordinates.

The **PtInRect** function determines whether a given point lies within a given rectangle. The point is in the rectangle if it lies on the left or top side or is completely within the rectangle.

The **IsRectEmpty** function determines whether the given rectangle is empty.

The **IntersectRect** function creates a new rectangle that is the intersection of two existing rectangles. The intersection is the largest rectangle contained in both existing rectangles. The intersection of two rectangles is shown in Figure 1.4:

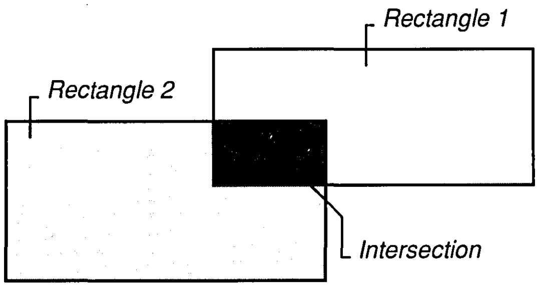


Figure 1.4 Intersection of Two Rectangles

The **UnionRect** function creates a new rectangle that is the union of two existing rectangles. The union is the smallest rectangle that contains both existing rectangles. The union of two rectangles is shown in Figure 1.5:

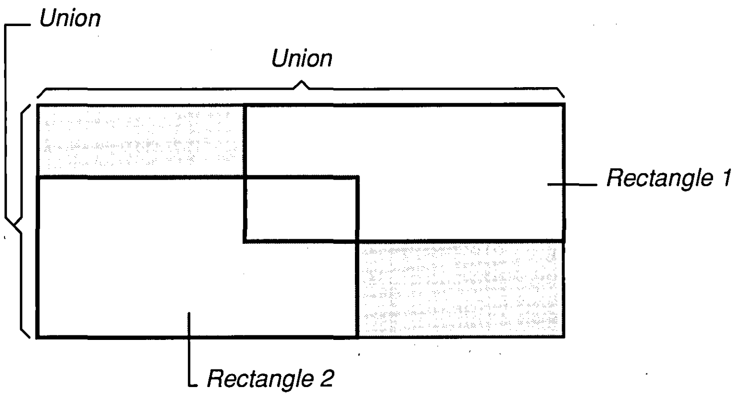


Figure 1.5 Union of Two Rectangles

For information about functions that draw ellipses and polygons, see Section 2.10, "Ellipse and Polygon Functions."

1.19 Summary

Window manager interface functions process messages, create, move, or alter a window, or create system output. For more information on topics related to window manager interface functions, see the following:

Topic	Reference
Function descriptions	*Reference, Volume 1*: Chapter 4, "Functions Directory"
Windows messages	*Reference, Volume 1*: Chapter 5, "Messages Overview," and Chapter 6, "Messages Directory"
Windows data types and structures	*Reference, Volume 2*: Chapter 7, "Data Types and Structures"
Using the Resource Compiler	*Reference, Volume 2*: Chapter 8, "Resource Script Statements"
	Tools: Chapter 3, "Compiling Resources: The Resource Compiler"
General information on Windows programming	*Guide to Programming*: Chapter 1, "An Overview of the Windows Environment"
Creating and managing a window	*Guide to Programming*: Chapter 2, "A Generic Windows Application"
Handling input	*Guide to Programming*: Chapter 4, "Keyboard and Mouse Input," and Chapter 6, "The Cursor, the Mouse, and the Keyboard"
Icons	*Guide to Programming*: Chapter 5, "Icons"
Menus	*Guide to Programming*: Chapter 7, "Menus"
Controls and dialog boxes	*Guide to Programming*: Chapter 8, "Controls," and Chapter 9, "Dialog Boxes"
Creating icons and cursors	*Tools*: Chapter 4, "Designing Images: SDKPaint"
Designing dialog boxes	*Tools*: Chapter 5, "Designing Dialog Boxes: The Dialog Editor"

Chapter 2 Graphics Device Interface Functions

This chapter describes the functions that perform device-independent graphics operations within a Windows application, including creating a wide variety of line, text, and bitmap output on many output devices. These functions constitute the Windows graphics device interface (GDI).

The chapter covers the following function categories:

- Device-context functions
- Drawing-tool functions
- Color-palette functions
- Drawing-attribute functions
- Mapping functions
- Coordinate functions
- Region functions
- Clipping functions
- Line-output functions
- Ellipse and polygon functions
- Bitmap functions
- Text functions
- Font functions
- Metafile functions
- Printer-control functions
- Printer-escape function
- Environment functions

2.1 Device-Context Functions

Device-context functions create, delete, and restore device contexts (DC). A device context is a link between a Windows application, a device driver, and an output device, such as a printer or plotter.

Figure 2.1 shows the flow of information from a Windows application through a device context and a device driver to an output device:

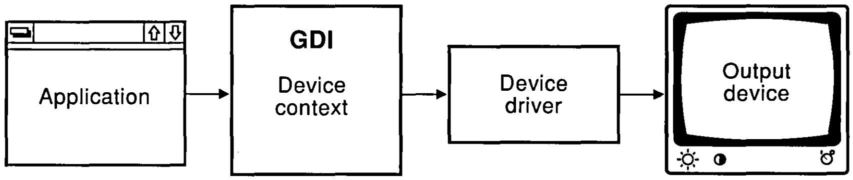


Figure 2.1 Information Flow to an Output Device

Any Windows application can use GDI functions to access an output device. GDI passes calls (which are device independent) from the application to the device driver. The device driver then translates the calls into device-dependent operations.

The following list briefly describes each device-context function:

Function	Description
CreateCompatibleDC	Creates a memory device context.
CreateDC	Creates a device context.
CreateIC	Creates an information context.
DeleteDC	Deletes a device context.
GetDCOrg	Retrieves the origin of a specified device context.
RestoreDC	Restores a device context.
SaveDC	Saves the current state of the device context.

2.1.1 Device-Context Attributes

Device-context attributes describe selected drawing objects (pens and brushes), the selected font and its color, the way in which objects are drawn (or mapped) to the device, the area on the device available for output (clipping region), and other important information. The data structure that contains these attributes is called the DC data block.

Table 2.1 lists the default device-context attributes and the GDI functions that affect or use these attributes:

Table 2.1 Default Device-Context Attributes and Related GDI Functions

Attribute	Default	GDI Functions
Background color	White	**SetBkColor**
Background mode	OPAQUE	**SetBkMode**
Bitmap	No default	**CreateBitmap** **CreateBitmapIndirect** **CreateCompatible-Bitmap** **SelectObject**
Brush	WHITE_BRUSH	**CreateBrushIndirect** **CreateDIBPatternBrush** **CreateHatchBrush** **CreatePatternBrush** **CreateSolidBrush** **SelectObject**
Brush origin	(0,0)	**SetBrushOrg** **UnrealizeObject**
Clipping region	Display surface	**ExcludeClipRect** **IntersectClipRect** **OffsetClipRgn** **SelectClipRgn**
Color palette	DEFAULT_PALETTE	**CreatePalette** **RealizePalette** **SelectPalette**
Current pen position	(0,0)	**MoveTo**
Drawing mode	R2_COPYPEN	**SetROP2**
Font	SYSTEM_FONT	**CreateFont** **CreateFontIndirect** **SelectObject**

Table 2.1 Default Device-Context Attributes and Related GDI Functions *(continued)*

Attribute	Default	GDI Functions
Intercharacter spacing	0	**SetTextCharacterExtra**
Mapping mode	MM_TEXT	**SetMapMode**
Pen	BLACK_PEN	**CreatePen** **CreatePenIndirect** **SelectObject**
Polygon-filling mode	ALTERNATE	**SetPolyFillMode**
Stretching mode	BLACKONWHITE	**SetStretchBltMode**
Text color	Black	**SetTextColor**
Viewport extent	(1,1)	**SetViewportExt**
Viewport origin	(0,0)	**SetViewportOrg**
Window extent	(1,1)	**SetWindowExt**
Window origin	(0,0)	**SetWindowOrg**

2.1.2 Saving a Device Context

Occasionally, it is necessary to save a device context so that the original attributes will be available at a later time. For example, a Windows application may need to save its original clipping region so that it can restore the client area's original state after a series of alterations occur. The **SaveDC** and **RestoreDC** functions make this possible.

2.1.3 Deleting a Device Context

The **DeleteDC** function deletes a device context and ensures that shared resources are not removed until the last context is deleted. The device driver is a shared resource.

2.1.4 Compatible Device Contexts

The **CreateCompatibleDC** function causes Windows to treat a portion of memory as a virtual device. This means that Windows prepares a device context that has the same attributes as the device for which it was created, but the device context has no connected output device. To use the compatible device context, the application creates a compatible bitmap and selects it into the device context. Any output it sends to the device is drawn in the selected bitmap. Since the device context is compatible with some actual device, the context of the bitmap can be copied directly to the actual device, or vice versa. This also means that the application can send output to memory (prior to sending it to the device). Note that the **CreateCompatibleDC** function works only for devices that have **BitBlt** capabilities.

2.1.5 Information Contexts

The **CreateIC** function creates an information context for a device. An information context is a device context with limited capabilities; it cannot be used to write to the device. An application uses an information context to gather information about the selected device. Information contexts are useful in large applications that require memory conservation.

By using an information context and the **GetDeviceCaps** function, you can obtain the following device information:

- Device technology
- Physical display size
- Color capabilities of the device
- Color-palette capabilities of the device
- Drawing objects available on the device
- Clipping capabilities of the device
- Raster capabilities of the device
- Curve-drawing capabilities of the device
- Line-drawing capabilities of the device
- Polygon-drawing capabilities of the device
- Text capabilities of the device

2.2 Drawing-Tool Functions

Drawing-tool functions create and delete the drawing tools that GDI uses when it creates output on a device or display surface. The following list briefly describes each drawing-tool function:

Function	Description
CreateBrushIndirect	Creates a logical brush.
CreateDIBPatternBrush	Creates a logical brush that has a pattern defined by a device-independent bitmap (DIB).
CreateHatchBrush	Creates a logical brush that has a hatched pattern.

Function	Description
CreatePatternBrush	Creates a logical brush that has a pattern defined by a memory bitmap.
CreatePen	Creates a logical pen.
CreatePenIndirect	Creates a logical pen.
CreateSolidBrush	Creates a logical brush.
DeleteObject	Deletes a logical pen, brush, font, bitmap, or region.
EnumObjects	Enumerates the available pens or brushes.
GetBrushOrg	Retrieves the current brush origin for a device context.
GetObject	Copies the bytes of logical data that define an object.
GetStockObject	Retrieves a handle to one of the predefined stock pens, brushes, fonts, or color palettes.
SelectObject	Selects an object as the current object.
SetBrushOrg	Sets the origin of all brushes selected into a given device context.
UnrealizeObject	Directs GDI to reset the origin of the given brush.

2.2.1 Drawing-Tool Uses

A Windows application can use any of three tools when it creates output: a bitmap, a brush, or a pen. An application can use the pen and brush together, outlining a region or object with the pen and filling the region's or object's interior with the brush. GDI allows the application to create pens with solid colors, bitmaps with solid or combination colors, and brushes with solid or combination colors. (The available colors and color combinations depend on the capabilities of the intended output device.)

Brushes

There are seven predefined brushes available in GDI; an application selects any one of them by using the **GetStockObject** function. The following list describes these brushes:

- Black
- Dark-Gray
- Gray
- Hollow

- Light-Gray
- Null
- White

There are six hatched brush patterns; an application can select any one of these patterns by using the **CreateHatchBrush** function. (A hatch line is a thin line that appears at regular intervals on a solid background.) The following list describes these hatch patterns:

- Backward Diagonal
- Cross
- Diagonal Cross
- Forward Diagonal
- Horizontal
- Vertical

Figure 2.2 shows each hatched brush pattern. A simple Windows application created this figure:

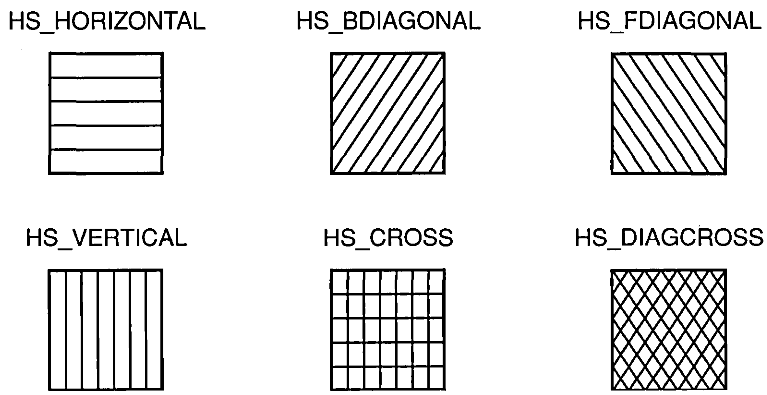


Figure 2.2 Hatched Brush Patterns

Pens

There are three predefined pens available in GDI; an application selects any one of them by using the **GetStockObject** function. The following list describes these pens:

- Black
- Null
- White

In addition to selecting a stock pen, an application creates an original pen by using the GDI **CreatePen** function. This function allows the application to select one of six pen styles, a pen width, and a pen color (if the device has color capabilities). The pen style can be solid, dashed, dotted, a combination of dots and dashes, or null. The pen width is the number of logical units GDI maps to a certain number of pixels (this number is dependent on the current mapping mode if the pen is selected into a device context). The pen color is an RGB color value.

Figure 2.3 shows a variety of pen patterns obtained from calls to the **CreatePen** function. A simple Windows application created this figure:

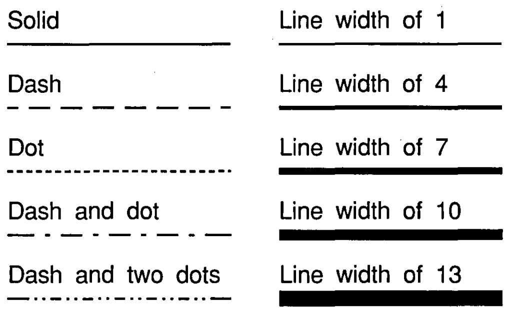


Figure 2.3 Pen Patterns

2.2.2 Color

Many of the GDI functions that create pens and brushes require that the calling application specify a color in the form of a **COLORREF** value. A **COLORREF** value specifies color in one of three ways:

- As an explicit RGB value
- As an index to a logical-palette entry
- As a palette-relative RGB value

The second and third methods require the application to create a logical palette. Section 2.3, "Color Palette Functions," describes Windows color palettes and the functions used by an application to exploit their capabilities.

An explicit RGB **COLORREF** value is a long integer that contains a red, a green, and a blue color field. The first (low-order) byte contains the red field, the second byte contains the green field, and the third byte contains the blue field; the fourth (high-order) byte must be zero. Each field specifies the intensity of the color; zero indicates the lowest intensity and 255 indicates the highest. For example, 0x00FF0000 specifies pure blue, and 0x0000FF00 specifies pure green. The RGB macro accepts values for the relative intensities of the three colors and returns an explicit RGB **COLORREF** value. When GDI receives the RGB value as a function parameter, it passes the RGB color value directly to the output device driver, which selects the closest available color on the device. The **GetNearestColor** function returns the closest logical color to a specified logical color that a given device can represent.

If the device is a plotter, the driver converts the RGB value to a single color that matches one of the pens on the device.

If the device uses color raster technology and the RGB value specifies a color for a pen, the driver will select a solid color. If the device uses color raster technology and the RGB value specifies a color for a brush, the driver will select from a variety of available color combinations. Since many color devices can display only a few colors, the actual color is simulated by "dithering," that is, mixing pixels of the colors which the display can actually render.

If the device is monochrome (black-and-white), the driver will select black, white, or a shade of gray, depending on the RGB value. If the sum of the RGB values is zero, the driver selects a black brush. If the sum of the RGB values is 765, the driver selects a white brush. If the sum of the RGB values is between zero and 765, the driver selects one of the gray patterns available.

The **GetRValue**, **GetGValue**, and **GetBValue** functions extract the values for red, green, and blue from an explicit RGB **COLORREF** value.

2.3 Color-Palette Functions

Many color graphic displays are capable of displaying a wide range of colors. In most cases, however, the actual number of colors which the display can render at any given time is more limited. For example, a display that is potentially able to produce over 262,000 different colors may be able to show only 256 of those colors at a time because of hardware limitations. In such cases, the display device often maintains a palette of colors; when an application requests a color that is not currently displayed, the display device adds the requested color to the palette. However, when the number of requested colors exceeds the maximum number for the device, it must replace an existing color with the requested color. As a result, if the total number of colors requested by one or more windows exceeds the number available on the display, many of the actual colors displayed will be incorrect.

Windows color palettes act as a buffer between color-intensive applications and the system, allowing an application to use as many colors as needed without interfering with its own color display or colors displayed by other windows. When a window has input focus, Windows ensures that the window will display all the colors it requests, up to the maximum number simultaneously available on the display, and displays additional colors by matching them to available colors. In addition, Windows matches the colors requested by inactive windows as closely as possible to the available colors. This significantly reduces undesirable changes in the colors displayed in inactive windows.

The following list briefly describes the functions an application calls to use color palettes:

Function	Description
AnimatePalette	Replaces entries in a logical palette; Windows maps the new entries into the system palette immediately.
CreatePalette	Creates a logical palette.
GetNearestPaletteIndex	Retrieves the index of a logical palette entry most nearly matching a specified RGB value.
GetPaletteEntries	Retrieves entries from a logical palette.
GetSystemPaletteEntries	Retrieves a range of palette entries from the system palette.
GetSystemPaletteUse	Determines whether an application has access to the full system palette.
RealizePalette	Maps entries in a logical palette to the system palette.
SelectPalette	Selects a logical palette into a device context.
SetPaletteEntries	Sets new palette entries in a logical palette; Windows does not map the new entries to the system palette until the application realizes the logical palette.
SetSystemPaletteUse	Allows an application to use the full system palette.
UpdateColors	Performs a pixel-by-pixel translation of each pixel's current color to the system palette. This allows an inactive window to correct its colors without redrawing its client area.

2.3.1 How Color Palettes Work

Color palettes provide a device-independent method for accessing the color capabilities of a display device by managing the device's physical (or system) palette, if one is available. Typically, devices that can display at least 256 colors use a physical palette.

An application employs the system palette by creating and using one or more *logical palettes*. Each entry in the palette contains a specific color. Then, instead of specifying an explicit value for a color when performing graphics operations, the application indicates which color is to be displayed by supplying an index into its logical palette.

Since more than one application can use logical palettes, it is possible that the total number of colors requested for display can exceed the capacity of the display device. Windows acts as a mediator among these applications.

When a window requests that its logical palette be given its requested colors (a process known as *realizing* its palette), Windows first exactly matches entries in the logical palette to current entries in the system palette.

If an exact match for a given logical-palette entry is not possible, Windows sets the entry in the logical palette into an unused entry in the system palette.

Finally, when all entries in the system palette have been used, Windows takes these logical palette entries that do not exactly match and matches them as closely as possible to entries already in the system palette. To further aid this color matching, Windows sets aside 20 static colors (called the "default palette") in the system palette to which it can match entries in a background palette.

Windows always satisfies the color requests of the foreground window first; this ensures that the active window will have the best color display possible. For the remaining windows, Windows satisfies the color requests of the window which most recently received input focus, the window which was active before that one, and so on.

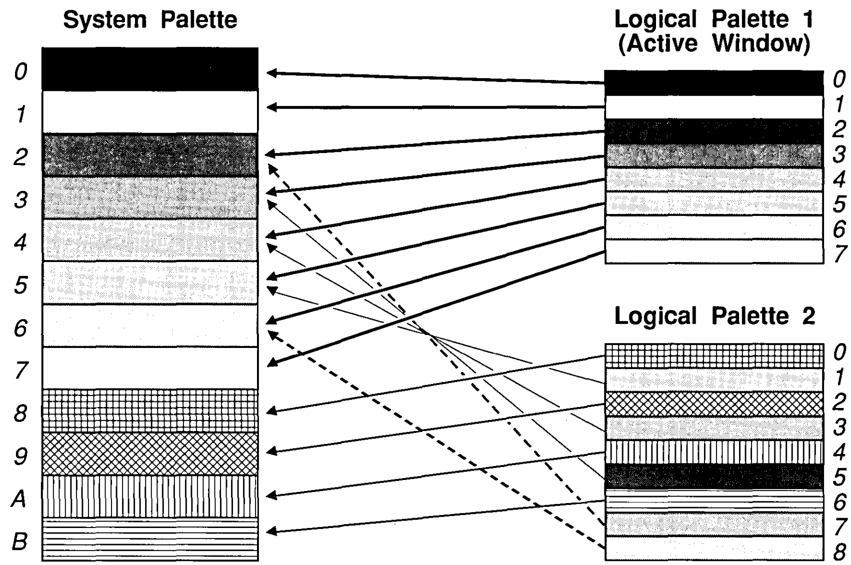


Figure 2.4 Palette Manager Color-Mapping Algorithm

Figure 2.4 illustrates this process. In this figure, a hypothetical display has a system palette capable of containing 12 colors. The application that created Logical Palette 1 owns the active window and was the first to realize its logical palette, which consists of 8 colors. Logical Palette 2 is owned by a window which realized its logical palette while it was inactive.

Because the active window was active when it realized its palette, Windows mapped all of the colors in Logical Palette 1 directly to the system palette.

Three of the colors (1, 3, and 5) in Logical Palette 2 are identical to colors in the system palette; to save space in the palette, then, Windows simply matched those colors to the existing system colors when the second application realized its palette. Colors 0, 2, 4, and 6 were not already in the system palette, however, and so Windows mapped those colors into the system palette.

Because the system palette is now full, Windows was not able to map the remaining two colors (which do not exactly match existing colors in the system palette) into the system palette. Instead, it matched them to the closest colors in the system palette.

2.3.2 Using a Color Palette

Before drawing to the display device using a color palette, an application must first create a logical palette by calling the **CreatePalette** function and then call **SelectPalette** to select the palette for the device context (DC) for the output device for which it will be used. An application *cannot* select a palette into a device context using the **SelectObject** function.

All functions which accept a color parameter accept an index to an entry in the logical palette. The palette-index specifier is a long integer value with the first bit in its high-order byte set to 1 and the palette index in the two low-order bytes. For example, 0x01000005 would specify the palette entry with an index of 5. The **PALETTEINDEX** macro accepts an integer value representing the index of a logical-palette entry and returns a palette-index **COLORREF** value which an application can use as a parameter for GDI functions that require a color.

An application can also specify a palette index indirectly by using a *palette-relative* RGB **COLORREF** value. If the target display device supports logical palettes, Windows matches the palette-relative RGB **COLORREF** value to the closest palette entry; if the target device does not support palettes, then the RGB value is used as though it were an explicit RGB **COLORREF** value. The palette-relative RGB **COLORREF** value is identical to an explicit RGB **COLORREF** value except that the second bit of the high-order byte is set to 1. For example, 0x02FF0000 would specify a palette-relative RGB **COLORREF** value for pure blue. The **PALETTERGB** macro accepts values for red, green and blue, and returns a palette-relative RGB **COLORREF** value which an application can use as a parameter for GDI functions that require a color.

If an application does specify an RGB value instead of a palette entry, Windows will use the closest matching color in the default palette of 20 static colors.

NOTE If the source and destination device contexts have selected and realized different palettes, the **BitBlt** function does not properly move bitmap bits to or from a memory device context. In this case, you must call the **GetDIBits** with the *wUsage* parameter set to DIB_RGB_COLORS to retrieve the bitmap bits from the source bitmap in a device-independent format. You then use the **SetDIBits** function to set the retrieved bits in the destination bitmap. This ensures that Windows will properly match colors between the two device contexts.

BitBlt can successfully move bitmap bits between two screen display contexts, even if they have selected and realized different palettes. The **StretchBlt** function properly moves bitmap bits between device contexts whether or not they use different palettes.

2.4 Drawing-Attribute Functions

Drawing-attribute functions affect the appearance of Windows output, which has four forms: line, brush, bitmap, and text. The following list describes each drawing-attribute function:

Function	Description
GetBkColor	Returns the current background color.
GetBkMode	Returns the current background mode.
GetPolyFillMode	Retrieves the current polygon-filling mode.
GetROP2	Retrieves the current drawing mode.
GetStretchBltMode	Retrieves the current stretching mode.
GetTextColor	Retrieves the current text color.
SetBkColor	Sets the background color.
SetBkMode	Sets the background mode.
SetPolyFillMode	Sets the polygon-filling mode.
SetROP2	Sets the current drawing mode.
SetStretchBltMode	Sets the stretching mode.
SetTextColor	Sets the text color.

2.4.1 Background Mode and Background Color

Line output can be solid or broken (dashed, dotted, or a combination of the two). If it is broken, the space between the breaks can be filled by setting the background mode to OPAQUE and selecting a color. By setting the background mode to TRANSPARENT, the space between breaks is left in its original state. The **SetBkMode** and **SetBkColor** functions accomplish this task.

Brush output is solid, patterned, or hatched. The space between hatch marks can be filled by setting the background mode to OPAQUE and selecting a color. When Windows creates brush output on a display, it combines the existing color on the display surface with the brush color to yield a new and final color; this is a binary raster operation. If the default raster operation is not appropriate, a new one is chosen by using the **SetROP2** function.

2.4.2 Stretch Mode

If an application copies a bitmap to a device and it is necessary to shrink or expand the bitmap before drawing, the effects of the **StretchBlt** and **StretchDIBits** functions can be controlled by calling **SetStretchBltMode** to set the current stretch mode for a device context. The stretch mode determines how lines eliminated from the bitmap are combined.

2.4.3 Text Color

The appearance of text output is limited only by the number of available fonts and the color capabilities of the output device. The **SetBkColor** function sets the color of the text background (the unused portion of each character's cell) and the **SetTextColor** function sets the color of the character itself.

2.5 Mapping Functions

Mapping functions alter and retrieve information about the GDI mapping modes. In order to maintain device independence, GDI creates output in a logical space and maps it to the display. The mapping mode defines the relationship between units in the logical space and pixels on a device. The following list briefly describes each mapping function:

Function	Description
GetMapMode	Retrieves the current mapping mode.
GetViewportExt	Retrieves a device context's viewport extents.
GetViewportOrg	Retrieves a device context's viewport origin.
GetWindowExt	Retrieves a device context's window extents.
GetWindowOrg	Retrieves a device context's window origin.
OffsetViewportOrg	Modifies a viewport origin.
OffsetWindowOrg	Modifies a window origin.
ScaleViewportExt	Modifies the viewport extents.
ScaleWindowExt	Modifies the window extents.
SetMapMode	Sets the mapping mode of a specified device context.
SetViewportExt	Sets a device context's viewport extents.
SetViewportOrg	Sets a device context's viewport origin.
SetWindowExt	Sets a device context's window extents.
SetWindowOrg	Sets a device context's window origin.

There are eight different mapping modes: MM_ANISOTROPIC, MM_HIENGLISH, MM_HIMETRIC, MM_ISOTROPIC, MM_LOENGLISH, MM_LOMETRIC, MM_TEXT, and MM_TWIPS. Each mode has a specific use in a Windows application. Table 2.2 summarizes the eight GDI mapping modes:

Table 2.2 GDI Mapping Modes

Mapping Mode	Intended Use
MM_ANISOTROPIC	Used in applications that map one logical unit to an arbitrary physical unit. The *x*- and *y*-axes are arbitrarily scaled.
MM_HIENGLISH	Used in applications that map one logical unit to 0.001 inch. Positive *y* extends upward.
MM_HIMETRIC	Used in applications that map one logical unit to 0.01 millimeter. Positive *y* extends upward.
MM_ISOTROPIC	Used in applications that map one logical unit to an arbitrary physical unit. One unit along the *x*-axis is always equal to one unit along the *y*-axis.
MM_LOENGLISH	Used in applications that map one logical unit to 0.01 inch. Positive *y* extends upward.
MM_LOMETRIC	Used in applications that map one logical unit to 0.1 millimeter. Positive *y* extends upward.
MM_TEXT	Used in applications that map one logical unit to one pixel. Positive *y* extends downward.
MM_TWIPS	Used in applications that map one logical unit to 1/1440 inch (1/20 of a printer's point). Positive *y* extends upward.

2.5.1 Constrained Mapping Modes

GDI classifies six of the mapping modes as constrained mapping modes: MM_HIENGLISH, MM_HIMETRIC, MM_LOENGLISH, MM_LOMETRIC, MM_TEXT, and MM_TWIPS. In each of these modes, one logical unit is mapped to a predefined physical unit. For instance, the MM_TEXT mode maps one logical unit to one device pixel, and the MM_LOENGLISH mode maps one logical unit to 0.01 inch on the device. These mapping modes are constrained because the scaling factor is fixed, so an application cannot change the number of logical units that Windows maps to a physical unit. Table 2.3 shows the number of logical units in various mapping modes that result in a certain physical unit:

Table 2.3 Logical/Physical Conversion Table

Mapping Mode	Logical Units	Physical Unit
MM_HIENGLISH	1000	1 inch
MM_HIMETRIC	100	1 millimeter
MM_LOENGLISH	100	1 inch
MM_LOMETRIC	10	1 millimeter
MM_TEXT	1	Device pixel
MM_TWIPS	1440	1 inch

2.5.2 Partially Constrained and Unconstrained Mapping Modes

The unconstrained mapping modes, MM_ISOTROPIC and MM_ANISOTROPIC, use two rectangular regions to derive a scaling factor and an orientation: the window and the viewport. The window lies within the logical-coordinate space and the viewport lies within the physical-coordinate space. Both possess an origin, an x-extent, and a y-extent. The origin may be any one of the four corners. The x-extent is the horizontal distance from the origin to its opposing corner. The y-extent is the vertical distance from the origin to its opposing corner. Windows creates a horizontal scaling factor by dividing the viewport's x-extent by the window's x-extent and creates a vertical scaling factor by dividing the viewport's y-extent by the window's y-extent. These scaling factors determine the number of logical units that Windows maps to a number of pixels. In addition to determining scaling factors, the window and viewport determine the orientation of an object. Windows always maps the window origin to the viewport origin, the window x-extent to the viewport x-extent, and the window y-extent to the viewport y-extent.

Partially Constrained Mapping Mode

An application creates output with equally scaled axes by using the MM_ISOTROPIC mapping mode. This means that Windows will map a symmetrical object (for example, a square or a circle) in the logical space as a symmetrical object in the physical space. In order to maintain this symmetry, GDI shrinks one of the viewport extents. The amount of shrinkage depends on the requested extents and the aspect ratio of the device. This mapping mode is called partially constrained because the application does not have complete control in altering the scaling factor.

Unconstrained Mapping Mode

An application can completely alter the horizontal and vertical scaling factors by using the MM_ANISOTROPIC mapping mode and setting the window and viewport extents to any value after selecting this mapping mode. Windows will not alter either scaling factor in this mode.

2.5.3 Transformation Equations

GDI uses the following equations to transform logical points to device points, and device points to logical points:

- Transforming logical points to device points:

  Dx = (Lx – xWO) × xVE/xWE + xVO
  Dy = (Ly – yWO) × yVE/yWE + yVO

- Transforming device points to logical points:

  Lx = (Dx – xVO) × xWE/xVE + xWO
  Ly = (Dy – yVO) × yWE/yVE + yWO

The following list describes the variables used in these transformation equations:

Variable	Description
xWO	Window origin *x*-coordinate
yWO	Window origin *y*-coordinate
xWE	Window extent *x*-coordinate
yWE	Window extent *y*-coordinate
xVO	Viewport origin *x*-coordinate
yVO	Viewport origin *y*-coordinate
xVE	Viewport extent *x*-coordinate
yVE	Viewport extent *y*-coordinate
Lx	Logical-coordinate system *x*-coordinate
Ly	Logical-coordinate system *y*-coordinate
Dx	Device *x*-coordinate
Dy	Device *y*-coordinate

The following four ratios are scaling factors:

xVE/xWE

yVE/yWE

xWE/xVE

yWE/yVE

They are used to determine the necessary stretching or compressing of logical units. The subtraction and addition of viewport and window origins is referred to as the translational component of the equation.

2.5.4 Example: MM_TEXT

The default mapping mode is MM_TEXT. In this mapping mode, one logical unit is mapped to one pixel on the device or display.

A simple Windows application created three rectangles as they appear in the logical and physical coordinate spaces when MM_TEXT is the mapping mode, as shown in Figure 2.5. The drawing on the left illustrates the logical space; the drawing on the right illustrates the device, or physical, space. The rectangles appear vertically elongated in the physical space because pixels on the chosen display are longer than they are wide. The rectangles appear to be upside-down because positive *y* extends downward in the physical-coordinate system.

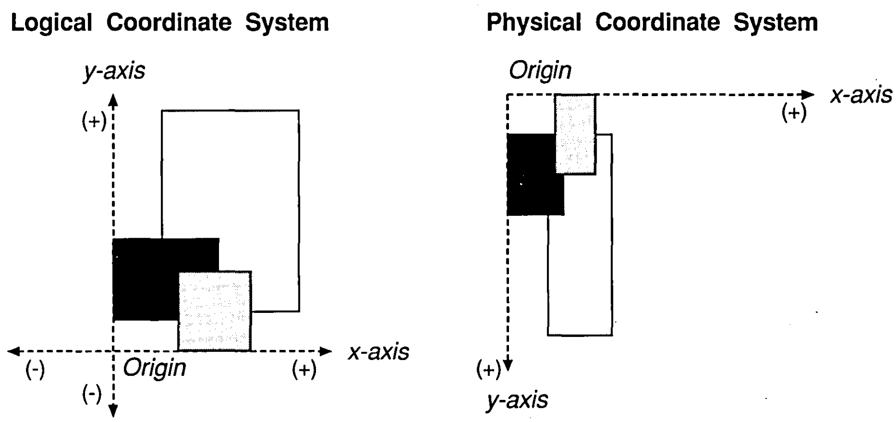


Figure 2.5 Mapping with MM_TEXT

2.5.5 Example: MM_LOENGLISH

A Windows application created three rectangles and mapped them from the logical space to the physical space by using the MM_LOENGLISH mapping mode, as shown in Figure 2.6. The drawing on the left illustrates how the rectangles appear in relation to the *x*- and *y*-axes in the logical coordinate system. The drawing on the right illustrates how the rectangles appear in relation to the *x*- and *y*-axes in the physical coordinate system.

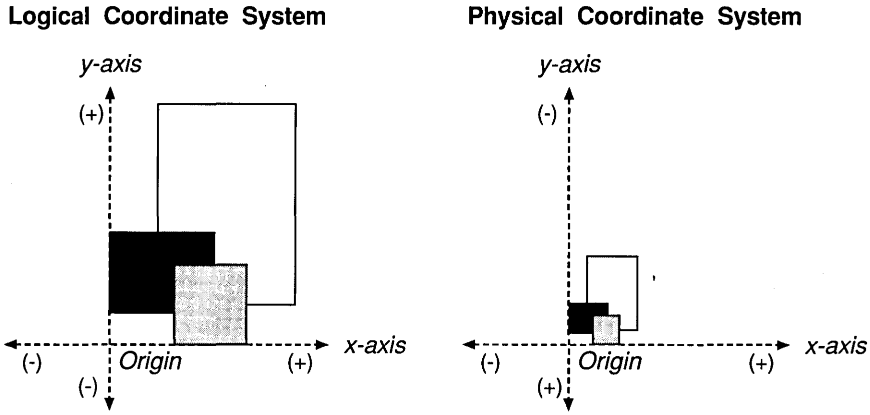


Figure 2.6 Mapping with MM_LOENGLISH

2.6 Coordinate Functions

Coordinate functions convert client coordinates to screen coordinates (or vice versa), and determine the location of a specific point. These functions are useful in graphics-intensive applications. The following list briefly describes each coordinate function:

Function	Description
ChildWindowFromPoint	Determines which child window contains a specified point.
ClientToScreen	Converts client coordinates into screen coordinates.
DPtoLP	Converts device points (that is, points relative to the window origin) into logical points.
LPtoDP	Converts logical points into device points.
ScreenToClient	Converts screen coordinates into client coordinates.
WindowFromPoint	Determines which window contains a specified point.

2.7 Region Functions

Region functions create, alter, and retrieve information about regions. A region is an elliptical or polygonal area within a window that can be filled with graphical output. An application uses these functions in conjunction with the clipping functions to create clipping regions. For more information about clipping functions, see Section 2.8, "Clipping Functions." The following list briefly describes each region function:

Function	Description
CombineRgn	Combines two existing regions into a new region.
CreateEllipticRgn	Creates an elliptical region.
CreateEllipticRgnIndirect	Creates an elliptical region.
CreatePolygonRgn	Creates a polygonal region.
CreatePolyPolygonRgn	Creates a region consisting of a series of closed polygons that are filled as though they were a single polygon.
CreateRectRgn	Creates a rectangular region.
CreateRectRgnIndirect	Creates a rectangular region.
CreateRoundRectRgn	Creates a rounded rectangular region.
EqualRgn	Determines whether two regions are identical.
FillRgn	Fills the given region with a brush pattern.
FrameRgn	Draws a border for a given region.
GetRgnBox	Retrieves the coordinates of the bounding rectangle of a region.
InvertRgn	Inverts the colors in a region.
OffsetRgn	Moves the given region.
PaintRgn	Fills the region with the selected brush pattern.
PtInRegion	Tests whether a point is within a region.

Function	Description
RectInRegion	Tests whether any part of a rectangle is within a region.
SetRectRgn	Creates a rectangular region.

2.8 Clipping Functions

Clipping functions create, test, and alter clipping regions. A clipping region is the portion of a window's client area where GDI creates output; any output sent to that portion of the client area which is outside the clipping region will not be visible. Clipping regions are useful in any Windows application that needs to save one part of the client area and simultaneously send output to another. The following list briefly describes each clipping function:

Function	Description
ExcludeClipRect	Excludes a rectangle from the clipping region.
GetClipBox	Copies the dimensions of a bounding rectangle.
IntersectClipRect	Forms the intersection of a clipping region and a rectangle.
OffsetClipRgn	Moves a clipping region.
PtVisible	Tests whether a point lies in a region.
RectVisible	Determines whether part of a rectangle lies in a region.
SelectClipRgn	Selects a clipping region.

2.9 Line-Output Functions

Line-output functions create simple and complex line output with the selected pen. The following list briefly describes each line-output function:

Function	Description
Arc	Draws an arc.
LineDDA	Computes successive points on a line.
LineTo	Draws a line with the selected pen.
MoveTo	Moves the current position to the specified point.
Polyline	Draws a set of line segments.

Figure 2.7 shows an arc created by using the **Arc** function. The upper portion of the illustration shows the arc as it would appear on a display; the lower portion shows the arc suspended in its bounding rectangle, which GDI uses to determine the size and shape of the arc:

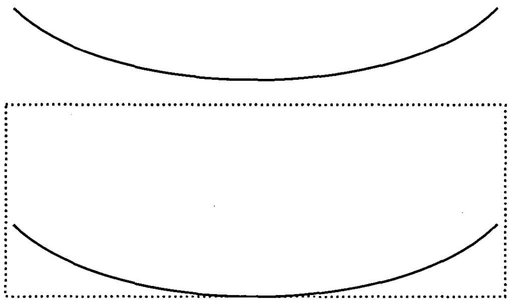


Figure 2.7 Arc and Its Bounding Rectangle

2.9.1 Function Coordinates

Line-output functions require coordinates in logical units, which GDI uses to draw a line in logical space. The use of logical units ensures device independence in Windows. GDI maps this line from the logical space to the physical space on the device. The number of logical units that GDI maps to a pixel depends on the current mapping mode. When GDI draws a line, it excludes the last specified point. For example, if the **LineTo** function is given the arguments (*X1*, *Y1*) and (*X2*, *Y2*), the line will be drawn from (*X1*, *Y1*) to (*X2* – 1, *Y2* – 1).

2.9.2 Pen Styles, Colors, Widths

If an application draws lines and does not create a new pen, GDI uses the default pen. This pen is black and is one pixel wide when the mapping mode is MM_TEXT. An application can create a new pen of a different width, style, and color by using the **CreatePen** function. The new color is dependent on the color capabilities of the output device. The new style can be solid, dotted, dashed, or a combination of dotted and dashed. Once an application creates a new pen, it can select it into a display context by using the **SelectObject** function.

Figure 2.8 shows simple line output created by the **LineTo** and **MoveTo** functions. The application created the rectangle on the left by using a styled pen and the rectangle on the right by using a solid pen:

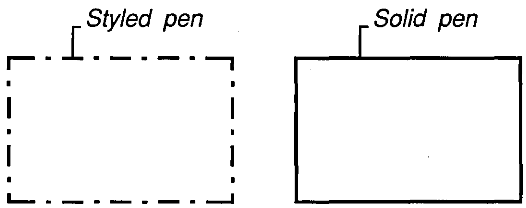


Figure 2.8 Styled-Pen and Solid-Pen Rectangles

2.10 Ellipse and Polygon Functions

Ellipse and polygon functions draw ellipses and polygons. GDI draws the perimeter of each object with the selected pen and fills the interior by using the selected brush. These functions are particularly useful in drawing and charting applications. The following list briefly describes each ellipse and polygon function:

Function	Description
Chord	Draws a chord.
DrawFocusRect	Draws a rectangle in the style used to indicate focus.
Ellipse	Draws an ellipse.
Pie	Draws a pie.
Polygon	Draws a polygon.
PolyPolygon	Draws a series of closed polygons that are filled as though they were a single polygon.
Rectangle	Draws a rectangle.
RoundRect	Draws a rounded rectangle.

2.10.1 Function Coordinates

Ellipse and polygon functions require coordinates in logical units, which GDI uses to determine the location and size of an object in logical space. The use of logical units ensures device independence in Windows. GDI uses a mapping function to map logical units to pixels on the device. The number of logical units that Windows maps to a pixel depends on the current mapping mode. The default mapping mode, MM_TEXT, maps one logical unit to one pixel.

When GDI draws a rectangle, it uses four arguments. The first two arguments specify the rectangle's upper-left corner. The last two arguments do not actually specify part of the rectangle; they specify the point adjacent to the lower-right corner. For example, if the first point is specified by (*X1*, *Y1*) and the second point is specified by ($X2$, $Y2$), the rectangle's upper-left corner will be ($X1$, $Y1$) and the lower-right corner will be ($X2 - 1$, $Y2 - 1$).

2.10.2 Bounding Rectangles

Instead of requiring a radius or circumference measurement, the **Chord**, **Ellipse**, and **Pie** functions use a bounding rectangle to define the size of the object they create. The bounding rectangle is hidden; GDI uses it only to describe the object's location and size.

For information about functions that alter or obtain information about rectangles in a window's client area, see Section 1.18, "Rectangle Functions."

2.11 Bitmap Functions

Bitmap functions display bitmaps. A bitmap is a matrix of memory bits that, when copied to a device, defines the color and pattern of a corresponding matrix of pixels on the device's display surface. Bitmaps are useful in drawing, charting, and word-processing applications because they let you prepare images in memory and then quickly copy them to the display. The following list briefly describes each bitmap function:

Function	Description
BitBlt	Copies a bitmap from a source to a destination device.
CreateBitmap	Creates a bitmap.
CreateBitmapIndirect	Creates a bitmap described in a data structure.
CreateCompatibleBitmap	Creates a bitmap that is compatible with a specified device.
CreateDiscardableBitmap	Creates a discardable bitmap that is compatible with a specified device.
ExtFloodFill	Fills the display surface within a border or over an area of a given color.
FloodFill	Fills the display surface within a border.
GetBitmapBits	Retrieves the bits in memory for a specific bitmap.
GetBitmapDimension	Retrieves the dimensions of a bitmap.

Function	Description
GetPixel	Retrieves the RGB value for a pixel.
LoadBitmap	Loads a bitmap from a resource file.
PatBlt	Creates a bit pattern.
SetBitmapBits	Sets the bits of a bitmap.
SetBitmapDimension	Sets the height and width of a bitmap.
SetPixel	Sets the RGB value for a pixel.
StretchBlt	Copies a bitmap from a source to a destination device (compresses or stretches, if necessary).

2.11.1 Bitmaps and Devices

The relationship between bitmap bits in memory and pixels on a device is device-dependent. On a monochrome device, the correspondence is usually one-to-one, where one bit in memory corresponds to one pixel on the device.

2.11.2 Device-Independent Bitmap Functions

Microsoft Windows version 3.0 provides a set of functions that define and manipulate color bitmaps which can be appropriately displayed on any device with a given resolution, regardless of the method by which the display represents color in memory. These functions translate a device-independent bitmap specification into the device-specific format used by the current display. The following is a list of these functions:

Function	Description
CreateDIBitmap	Creates a device-specific memory bitmap from a device-independent bitmap (DIB) specification and optionally initializes bits in the bitmap. This function is similar to **CreateBitmap**.
GetDIBits	Retrieves the bits in memory for a specific bitmap in device-independent form. This function is similar to **GetBitmapBits**.
SetDIBits	Sets a memory bitmap's bits from a DIB. This function is similar to **SetBitmapBits**.
SetDIBitsToDevice	Sets bits on a device surface directly from a DIB.

Function	Description
StretchDIBits	Moves a device-independent bitmap (DIB) from a source rectangle into a destination rectangle, stretching or compressing the bitmap as required.

A device-independent bitmap specification consists of two parts:

1. A **BITMAPINFO** data structure that defines the format of the bitmap and optionally supplies a table of colors used by the bitmap
2. An array of bytes that contain the bitmap bit values

Depending on the values contained in the bitmap information data structure, the bitmap bit values can specify explicit color (RGB) values or indexes into the color table. In addition, the color table can consist of indexes into the currently realized logical palette instead of explicit RGB color values. It is important to note that the coordinate-system origin for DIBs is the lower-left corner, not the Windows default upper-left corner.

2.12 Text Functions

Text functions retrieve text information, alter text alignment, alter text justification, and write text on a device or display surface. GDI uses the current font for text output. The following list briefly describes each text function:

Function	Description
ExtTextOut	Writes a character string, within a rectangular region, using the currently selected font. The rectangular region can be opaque (filled with the current background color) and it can be a clipping region.
GetTabbedTextExtent	Computes the width and height of a line of text containing tab characters.
GetTextAlign	Returns a mask of the text alignment flags.
GetTextExtent	Uses the current font to compute the width and height of text.
GetTextFace	Copies the current font name to a buffer.
GetTextMetrics	Fills the buffer with metrics for the selected font.
SetTextAlign	Positions a string of text on a display or device.
SetTextJustification	Justifies a text line.

Function	Description
TabbedTextOut	Writes a character string with expanded tabs, using the current font.
TextOut	Writes a character string using the current font.

2.13 Font Functions

Font functions select, create, remove, and retrieve information about fonts. A font is a subset of a particular typeface, which is a set of characters that share a similar fundamental design.

The following list briefly describes each font function:

Function	Description
AddFontResource	Adds a font resource in the specified file to the system font table.
CreateFont	Creates a logical font that has the specified characteristics.
CreateFontIndirect	Creates a logical font that has the specified characteristics.
EnumFonts	Enumerates the fonts available on a given device.
GetCharWidth	Retrieves the widths of individual characters.
RemoveFontResource	Removes a font resource from the font table.
SetMapperFlags	Alters the algorithm the font mapper uses.

A font family is a group of typefaces that have similar stroke-width and serif characteristics. A typeface is a set of characters (letters, numerals, punctuation marks, symbols) that share a common design. Font characters share very specific characteristics, such as point size and weight.

Note that the terms GDI uses to describe fonts, typefaces, and families of fonts do not necessarily correspond to traditional typographic terms.

The Helv typeface is an example of a familiar typeface. Available fonts within this typeface include 8-point Helv bold and 10-point Helv italic.

Figure 2.9 shows several fonts from the Helv and Courier typefaces:

This is a line of 12 point Helv.

This is a line of 12 point Helv bold.

This is a line of 12 point Helv italic.

`This is a line of 12 point Courier.`

`This is a line of 12 point Courier bold.`

`This is a line of 12 point Courier italic.`

Figure 2.9 Fonts from Two Typefaces

2.13.1 Font Family

GDI organizes fonts by family; each family consists of typefaces and fonts that share a common design. The families are divided by stroke width and serif characteristics. The term stroke, which means a horizontal or vertical line, comes from handwritten characters composed of one or more pen strokes. The horizontal stroke is called a cross-stroke. The main vertical line is called a stem. Figure 2.10 shows a lowercase *f* composed of a cross-stroke and a stem with a loop at the top:

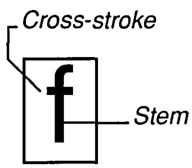


Figure 2.10 Cross-Stroke and Stem

Serifs are short cross-lines drawn at the ends of the main strokes of a letter. If a typeface does not have serifs, it is generally called a sans-serif (without serif) typeface. Figure 2.11 shows serifs:

Figure 2.11 Serifs

GDI uses five distinct family names to categorize typefaces and fonts. A sixth name is used for generic cases. Note that GDI's family names do not correspond to traditional typographic categories. Table 2.4 lists the font-family names and briefly describes each family:

Table 2.4 Font Families

Name	Description
Dontcare	Generic family name. Used when information about a font does not exist or does not matter.
Decorative	Novelty fonts.
Modern	Constant stroke width (fixed-pitch), with or without serifs. Fixed-pitch fonts are usually modern.
Roman	Variable stroke width (proportionally spaced), with serifs.
Script	Designed to look like handwriting.
Swiss	Variable stroke width (proportionally spaced), without serifs.

2.13.2 Character Cells

A character is the basic element in a font. In GDI, each character is contained within a rectangular region known as a character cell. This rectangular region consists of a specific number of rows and columns, and possesses six points of measurement: ascent, baseline, descent, height, origin, and width. The following list describes these measurements:

Measurement	Description
Ascent	Specifies the distance in character-cell rows from the character-cell baseline to the top of the character cell.
Baseline	Serves as the base on which all characters stand (some lowercase letters have descenders, such as the tail of the *g* or *y*, that descend below the baseline).
Descent	Specifies the distance in character-cell rows from the character-cell baseline to the bottom of the character cell.
Height	Specifies the height of a character-cell row.

Measurement	Description
Origin	Used as a point of reference when the character is written on a device or a display surface. The origin is the upper-left corner of the character cell.
Width	Specifies the width of a character-cell column.

Figure 2.12 shows a character cell that contains an uppercase *A*. The baseline appears at the top of the second row. Note that the uppercase *A* uses the baseline as its starting point. Also note that the width and height values refer to the character-cell width and height, not the width and height of the individual character:

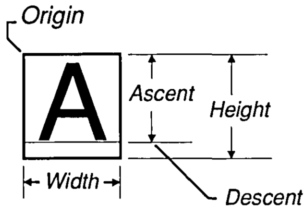


Figure 2.12 Character-Cell Dimensions

2.13.3 Altering Characters

Characters exist in many sizes and shapes. The following sections describe how characters are altered in GDI to produce a particular font.

Italic

For an italic font, GDI skews the characters so that they appear slanted. When italicized, the base of the character remains intact while the upper portion shifts to the right. The greatest amount of shifting occurs at the top of the character, the least amount at the base. Figure 2.13 shows characters before and after being italicized:

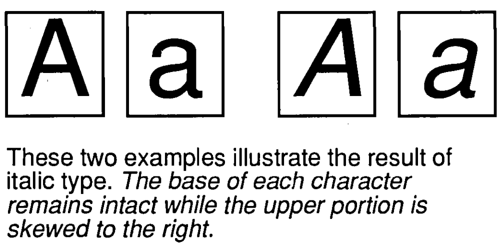
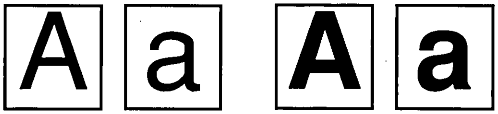


Figure 2.13 Normal and Italic Characters

Bold

A font is made bold by increasing its weight, which refers to the thickness of the lines or strokes that compose a character. Fonts with a heavy weight are referred to as bold. Figure 2.14 shows normal and bold characters:

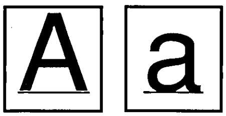


These two examples illustrate the result of varying font weight. **A heavier weight gives you a bolder font.**

Figure 2.14 Normal and Bold Characters

Underline

An underline font has a line under each character. When a character is underlined, a solid line appears directly below the baseline of the character cell. Figure 2.15 shows underlined characters:



This font is underlined.
A solid line is drawn
below the baseline of
each character cell.

Figure 2.15 Underlined Characters

Strikeout

A strikeout font has a solid horizontal line drawn through each character. The position of this line within each character cell is constant for a given font. Figure 2.16 shows characters that are struck out:

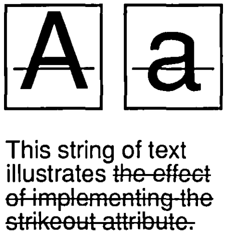


Figure 2.16 Strikeout Characters

2.13.4 Leading

Leading is the distance from baseline to baseline of two adjacent rows of text. When font designers develop a font, they specify that a given amount of space should appear between rows. The addition of this space ensures that a character is not obscured by part of another character in an adjacent row. There are two ways of adding this additional space: by inserting it within the character cells of a font (internal leading) or by inserting it between rows of text as they are printed on a device (external leading).

Internal Leading

Internal leading refers to the space inserted within character cells of a particular font. Only marks such as accents, umlauts, and tildes in foreign character sets appear within the space allocated for internal leading. Figure 2.17 shows two rows of text that use internal leading:

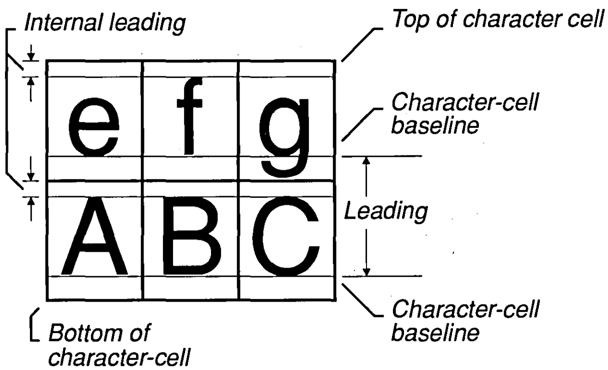


Figure 2.17 Internal Leading

External Leading

External leading is space inserted between the top and bottom of character cells in adjacent rows of text. The font designer must specify the amount of external leading necessary to produce easily readable text from a particular font. External leading is not built into a font; you must add it before you print text on a device. Figure 2.18 shows external leading:

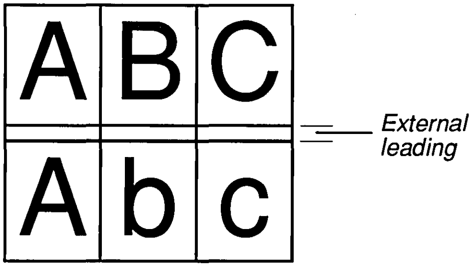


Figure 2.18 External Leading

2.13.5 Character Set

All fonts use a character set. A character set contains punctuation marks, numerals, uppercase and lowercase letters, and all other printable characters. The designer of a character set assigns a numeric value to each element in the set. You use this number to access an element within the set.

Most character sets used in Windows are supersets of the U.S. ASCII character set, which defines characters for the 96 numeric values from 32 to 127. There are four major groups of character sets:

- ANSI
- OEM
- Symbol
- Vendor specific

ANSI Character Set

The ANSI character set is the most commonly used character set. The blank character is the first character in the ANSI character set. It has a hexadecimal value of 0x20, which is equivalent to the decimal value 32. The last character in the ANSI character set has a hexadecimal value of 0xFF, which is equivalent to the decimal value 255.

Many fonts specify a default character. Whenever a request is made for a character not in the set, this default character is given. Most fonts using the ANSI character set specify the period (.) as the default character. The hexadecimal value for the period is 0x2E, or decimal 46 in the ANSI character set.

Fonts use a break character to separate words and justify text. Most fonts using the ANSI character set specify the blank character, whose hexadecimal value is 0x20, decimal 32.

OEM Character Set

Windows supports a second character set, referred to as the OEM character set. This is generally the character set used internally by DOS for screen display. Characters 32 to 127 of the OEM set are usually identical to the same characters in the U.S. ASCII set, which are also in the ANSI set. The remaining characters in the OEM set (0 to 31, and 128 to 255) correspond to the characters which may be shown on the computer's DOS display, and generally differ from ANSI characters.

Symbol Character Set

The symbol character set contains special characters typically used to represent mathematical and scientific formulas.

Vendor-Specific Character Sets

Many printers and other output devices contain fonts based on character sets which differ from the ANSI and OEM sets, such as the EBCDIC character set. In such cases, the printer driver must translate from the ANSI character set to one or more of the sets provided by the printer or other device.

2.13.6 Pitch

The term pitch traditionally refers to the number of characters from a particular font that will fit in a single inch. GDI, however, uses this term differently. The term fixed-pitch refers to a font whose character-cell size is constant for each character. The term variable-pitch refers to a font whose character cells vary in size, depending on the actual width of the characters.

Average Character Width

Variable-pitch fonts use the average character width to specify the average width of character cells in the font. Since there is no variance in character-cell width for fixed-pitch fonts, the average character width specifies the character width of any character in the fixed-pitch font.

Maximum Character Width

Variable-pitch fonts use the maximum character width to specify the maximum width of any character cell in the font. Since there is no variance in character width for fixed-pitch fonts, the maximum character width is equivalent to the average character width in the fixed-pitch font.

Digitized Aspect

When raster fonts are created, they are designed with one particular aspect ratio in mind. The aspect ratio is the ratio of the width and height of a device's pixel. GDI maintains a record of the ideal *x*-aspect and *y*-aspect for individual fonts. The ideal *x*-aspect is the width value from the aspect ratio of the device. The ideal *y*-aspect is the height value from the aspect ratio of the device. These values are called the digitized aspects for *x* and *y*. The **GetAspectRatioFilter** function retrieves the setting for the current aspect-ratio filter. Windows provides a special filter, the aspect-ratio filter, to select fonts designed for a particular aspect ratio from all of the available fonts. The filter uses the aspect ratio specified by the **SetMapperFlags** function.

Overhang

When a particular font is not available on a device, GDI sometimes synthesizes that font. The process of synthesizing may add width or height to an existing font. Whenever GDI synthesizes an italic or bold font from a normal font, extra columns are added to individual character cells in that font. The difference in width (the extra columns) between a string created with the normal font and a string created with the synthesized font is called the overhang.

2.13.7 Selecting Fonts with GDI

GDI maintains a collection of fonts from different typefaces. In addition to this collection, some devices maintain a collection of hardware fonts in their ROM. GDI lets you describe a font and then selects the closest matching available font from your description.

GDI requires you to describe the font you want to use to create text. The font you describe is a logical font (it may or may not actually exist). GDI compares this logical font to the available physical fonts and selects the closest match.

The process of selecting the physical font that bears the closest resemblance to the specified logical font is known as font mapping. GDI also maintains a font table. Each entry in the font table describes a physical font and its attributes. Included in each entry is a pointer to a corresponding font resource. Figure 2.19 shows a font table that contains fonts X, Y, and Z:

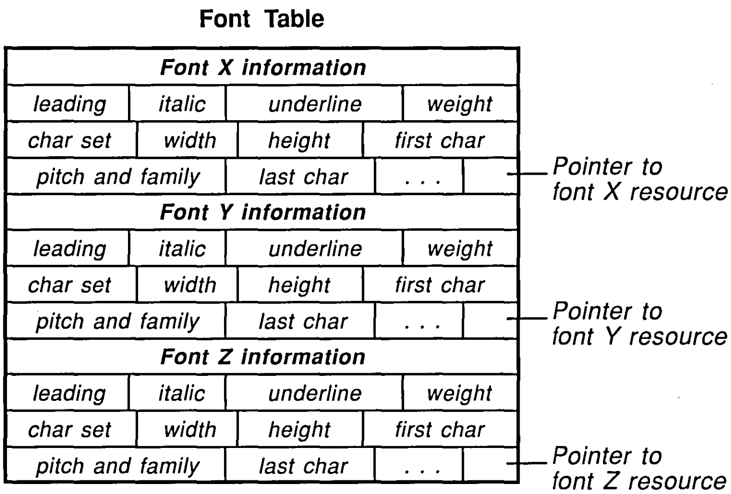


Figure 2.19 A GDI Font Table

Font-Mapping Scheme

GDI cannot guarantee that a physical font exists that exactly matches a requested logical font, so GDI attempts to pick a font that has the fewest differences from the requested logical font. Since fonts have many different attributes, the GDI font mapper assigns penalties to physical fonts whose characteristics do not match the characteristics of the specified logical font. The physical font with the fewest penalties assigned is the one that GDI selects.

To begin the mapping, GDI transforms the requested height and width of the logical font to device units. This transformation depends on the current mapping mode and window and viewport extents. GDI then asks the device to realize the physical font. A device can realize a font if it can create it or a font very close to it.

If the device can realized a physical font, GDI compares this font with its own set of fonts. If GDI has a font that more closely matches the logical font, GDI uses it. But if the device signals that it can take device-realized fonts only, GDI uses the realized font.

If the device cannot realize a font, GDI searches its own fonts for a match.

To determine how good a match a given physical font is to the requested logical font, the mapper takes the logical font and compares it one attribute at a time with each physical font in the system.

Table 2.5 lists the characteristics that are penalized by GDI's font mapper. The characteristics are grouped according to penalty weights, with the heaviest penalty assigned to the CharSet characteristic and the lightest penalty assigned to the Weight, Slant, Underline, and StrikeOut characteristics.

Table 2.5 Font-Mapping Characteristics

Characteristic	Penalty Scheme	Penalty Weight
CharSet	If the character set does not match, the candidate font is penalized heavily. Fonts with the wrong character set are very rarely selected as the physical font. There is no default character set. This means a logical font must alway specify the desired set.	4
Pitch	The wrong pitch is penalized heavily. If the requested pitch is fixed, a wrong pitch is assessed a greater penalty since an application that handles fixed pitches may not be able to handle variable-pitch fonts.	3
Family	If the font families do not match, the candidate font is penalized heavily. If a default font family is requested, no penalties are assessed.	3
FaceName	If the font typeface names do not match, the candidate font is penalized heavily. If a default font facename is requested, no penalties are assessed.	3
Height	The wrong height is penalized. GDI always chooses or synthesizes a shorter font if the exact height is not available. GDI can synthesize a font by expanding a font's character bitmaps by an integer multiple. GDI will expand a font up to eight times. If a default height is requested, GDI arbitrarily searches for a twelve-point font.	2
Width	The wrong width is penalized. GDI always chooses or synthesizes a narrower font if the exact width is not available. If a default width is requested, GDI assesses a penalty for any difference between the aspect ratio of the device and the aspect ratio of the font. The mapper can give unexpected results if there are no fonts for the given aspect ratio.	2
Weight	Although GDI can synthesize bold, an actual bold font is preferred. The mapper penalizes for synthesizing.	1
Slant	Although GDI can synthesize italics, an actual italic font is preferred. The mapper penalizes for synthesizing.	1

Table 2.5 Font-Mapping Characteristics *(continued)*

Characteristic	Penalty Scheme	Penalty Weight
Underline	Although GDI can synthesize underlining, an actual underline font is preferred. The mapper penalizes for synthesizing.	1
StrikeOut	Although GDI can synthesize strikeouts, an actual strikeout font is preferred. The mapper penalizes for synthesizing.	1

If GDI synthesizes a font, the mapper assesses a penalty that depends on the number of times the font was replicated. Furthermore, a penalty is added if the font was synthesized in both directions and the synthesizing was uneven, that is, if the font was stretched more in one direction than the other.

When the mapper has compared all the fonts in the system, it picks the one with the smallest penalty. The application should retrieve the metrics of the font to find out the characteristics of the font it received.

The penalty weights listed in Table 2.5 are the default penalties used by GDI.

Example of Font Selection

For the purpose of this example, assume that the system font table lists only the three fonts shown in Figure 2.19, "A GDI Font Table," fonts X, Y, and Z. Suppose you need to use a specific font, font Q, to create text on an output device. You will need to describe font Q so that GDI can choose the physical font (X, Y, or Z) that bears the closest resemblance to Q.

To describe font Q, you use the **CreateFont** or **CreateFontIndirect** GDI function. These functions create a logical font which is a description of the desired physical font.

Use the **SelectObject** function to select the physical font that most closely matches logical font Q. (The **SelectObject** function requires that you pass a handle to font Q.) Once a call to the **SelectObject** function occurs, GDI will initiate the selection process.

Table 2.6 shows the physical fonts in the font table and the penalties that GDI assigns to each as it tries to find a font that will match font Q. The left column shows the font attributes that GDI compares; the second column gives the attributes of font Q, the desired font. The attributes of fonts X, Y, and Z—the fonts that are actually in the system font table—are followed by the penalty values that GDI gives to each one. The bottom row of the table gives the penalty totals for each font:

Table 2.6 Sample Font Selection Ratings

	Desired	Available Fonts/Penalty Score					
Attributes	**Q**	**X**		**Y**		**Z**	
CharSet	ANSI	OEM	4	OEM	4	ANSI	0
Pitch	Fixed	Variable	3	Fixed	0	Variable	3
Family	Roman	Modern	3	Roman	0	Modern	3
FaceName	Tms Rmn	Pica	3	Tms Rmn	0	Elite	3
Height	8	10	2	10	2	8	0
Width	4	6	2	6	2	4	0
Slant	None	None	0	None	0	None	0
Underline	None	None	0	None	0	None	0
StrikeOut	None	None	0	None	0	None	0
Penalty Total			17		8		9

The penalty totals show that font Y has the lowest penalty score and therefore resembles font Q most closely. In this example, GDI would select font Y as the physical font on the output device.

2.13.8 Font Files and Font Resources

GDI stores information about the physical font in font files. The font file consists of a header and a bitmap. The font-file header contains a detailed description of the font. If the font file is a raster file, the font-file bitmap contains actual representations of the font characters. If the font file is a vector file, the font-file bitmap contains character strokes for the font characters. A font resource is a collection of one or more of these physical-font files.

2.14 Metafile Functions

Metafile functions close, copy, create, delete, retrieve, play, and return information about metafiles. A metafile is a collection of GDI commands that creates desired text or images.

Metafiles provide a convenient method of storing graphics commands that create text or images. Metafiles are especially useful in applications that use specific text or a particular image repeatedly. They are also device-independent; by creating text or images with GDI commands and then placing the commands in a metafile, an application can re-create the text or images repeatedly on a variety of devices. Metafiles are also useful in applications that need to pass graphics information to other applications.

The following list briefly describes each metafile function:

Function	Description
CloseMetaFile	Closes a metafile and creates a metafile handle.
CopyMetaFile	Copies a source metafile to a file.
CreateMetaFile	Creates a metafile display context.
DeleteMetaFile	Deletes a metafile from memory.
EnumMetaFile	Enumerates the GDI calls within a metafile.
GetMetaFile	Creates a handle to a metafile.
GetMetaFileBits	Stores a metafile as a collection of bits in a global memory block.
PlayMetaFile	Plays the contents of a specified metafile.
PlayMetaFileRecord	Plays a metafile record.
SetMetaFileBits	Creates a memory metafile.

2.14.1 Creating a Metafile

A Windows application must create a metafile in a special device context. It cannot use the device contexts that the **CreateDC** or **GetDC** functions return; instead, it must use the device context that the **CreateMetaFile** function returns.

Windows allows an application to use a subset of the GDI functions to create a metafile. This subset is the set of all GDI functions that create output (it is not necessary to use those functions that provide state information, such as the **GetDeviceCaps** or **GetEnvironment** functions). The following is a list of GDI functions an application can use in a metafile:

AnimatePalette	**OffsetViewportOrg**	**SetDIBitsToDevice**
Arc	**OffsetWindowOrg**	**SetMapMode**
BitBlt	**PatBlt**	**SetMapperFlags**
Chord	**Pie**	**SetPixel**
CreateBrushIndirect	**Polygon**	**SetPolyFillMode**
CreateDIBPatternBrush	**Polyline**	**SetROP2**
CreateFontIndirect	**PolyPolygon**	**SetStretchBltMode**
CreatePatternBrush	**RealizePalette**	**SetTextAlign**
CreatePenIndirect	**Rectangle**	**SetTextCharExtra**
CreateRegion	**ResizePalette**	**SetTextColor**
DrawText	**RestoreDC**	**SetTextJustification**
Ellipse	**RoundRect**	**SetViewportExt**
Escape	**SaveDC**	**SetViewportOrg**
ExcludeClipRect	**ScaleViewportExt**	**SetWindowExt**
ExtTextOut	**ScaleWindowExt**	**SetWindowOrg**
FloodFill	**SelectClipRegion**	**StretchBlt**
IntersectClipRect	**SelectObject**	**StretchDIBits**
LineTo	**SelectPalette**	**TextOut**
MoveTo	**SetBkColor**	
OffsetClipRgn	**SetBkMode**	

To create output with a metafile, an application must follow four steps:

1. Create a special device context by using the **CreateMetaFile** function.
2. Send GDI commands to the metafile by using the special device context.
3. Close the metafile by calling the **CloseMetaFile** function. This function returns a metafile handle.
4. Display the image or text on a device by using the **PlayMetaFile** function, passing to the function the metafile handle obtained from **CloseMetaFile** and a device-context handle for the device to which the metafile is to be played.

The device context which **CreateMetaFile** creates does not have default attributes of its own. Whatever device-context attributes are in effect for the output device when an application plays a metafile will be the defaults for the metafile. The metafile can change these attributes while it is playing. If the application needs to retain the same device-context attributes after the metafile has finished playing, it should save the output device context by calling the **SaveDC** function before calling **PlayMetaFile**. Then, when **PlayMetaFile** returns, the application can call the **RestoreDC** function (with –1 as the *nSavedDC* parameter) to restore the original device-context attributes.

Although the maximum size of a metafile is 2^{32} bytes or records, the actual size of a metafile is limited by the amount of memory or disk space available.

2.14.2 Storing a Metafile in Memory or on Disk

An application can store a metafile in system memory or in a disk file.

To store the metafile in memory, an application calls **CreateMetafile** and passes NULL as the function parameter.

There are two ways of storing a metafile in a disk file:

- When the application calls **CreateMetaFile** to open a metafile, it passes a filename as the function parameter; the metafile will then be recorded in a disk file.
- After the application has created a metafile in memory, it calls the **Copy-MetaFile** function. This function accepts the handle of a memory metafile and the filename of the disk file which is to save the metafile.

The **GetMetaFile** function opens a metafile stored in a disk file and makes it available for replay or modification. This function accepts the filename of a metafile stored on disk and returns a metafile handle.

2.14.3 Deleting a Metafile

An application frees the memory which Windows uses to store the metafile by calling the **DeleteMetafile** function. This function removes a metafile from memory and invalidates its handle. It has no effect on disk files.

2.14.4 Changing How Windows Plays a Metafile

A metafile does not have to be played back in its entirety or exactly in the form in which it was recorded. An application can use the **EnumMetaFile** function to locate a specific metafile record. **EnumMetaFile** calls an application-supplied callback function and passes it the following:

- The metafile device context
- A pointer to the metafile handle table
- A pointer to a metafile record
- The number of associated objects with handles in the handle table
- A pointer to application-supplied data

The callback function can then use this information to play a single record, to query it, copy it, or modify it. The **PlayMetaFileRecord** function plays a single metafile record.

Chapter 9, "File Formats," in *Reference, Volume 2*, shows the formats of the various metafile records and describes their contents.

When Windows plays or enumerates the records in a metafile, it identifies each object with an index into a handle table. Functions that select objects (such as **SelectObject** and **SelectPalette**) identify the object by means of the object handle which the application passes to the function.

Objects are added to the table in the order in which they are created. For example, if a brush is the first object created in a metafile, the brush is given index zero. If the second object is a pen, it is given index 1, and so on. See the description of the **HANDLETABLE** data structure in Chapter 7, "Data Types and Structures," in *Reference, Volume 2*, for information on the format of the handle table.

2.15 Printer-Control Functions

Printer-control functions retrieve information about a printer and modify its initialization state. The printer driver, rather than GDI itself, provides these functions. The following list briefly describes each printer-control function:

Function	Description
DeviceCapabilities	Retrieves capabilities of a printer device driver.
DeviceMode	Sets the current printing modes for a device by prompting the user with a dialog box.
ExtDeviceMode	Retrieves or modifies device initialization information for a given printer driver or displays a driver-supplied dialog box for configuring the driver.

2.16 Printer-Escape Function

The **Escape** function allows an application to access facilities of a particular device that are not directly available through GDI. The *nEscape* parameter of this function specifies the escape function to be performed. When an application calls **Escape** for a printer device context, the escape functions regulate the flow of printer output from Windows applications, retrieve information about a printer, and alter the settings of a printer.

2.16.1 Creating Output on a Printer

Windows applications use only the standard Windows functions to access system memory, the output device, the keyboard, and the mouse. Each application interacts with the user through one or more windows that are created and maintained by the user. GDI assists an application in creating output by passing device-independent function calls from the application to the device driver. The device driver first translates these device-independent function calls into device-dependent operations that create images on a device's display surface, and then sends them to Print Manager (the spooler). Print Manager serves two purposes: it collects translated commands from one application and stores them in a corresponding job, and it passes a complete job to the device for output. Figure 2.20 shows the path of output from a Windows application to a device:

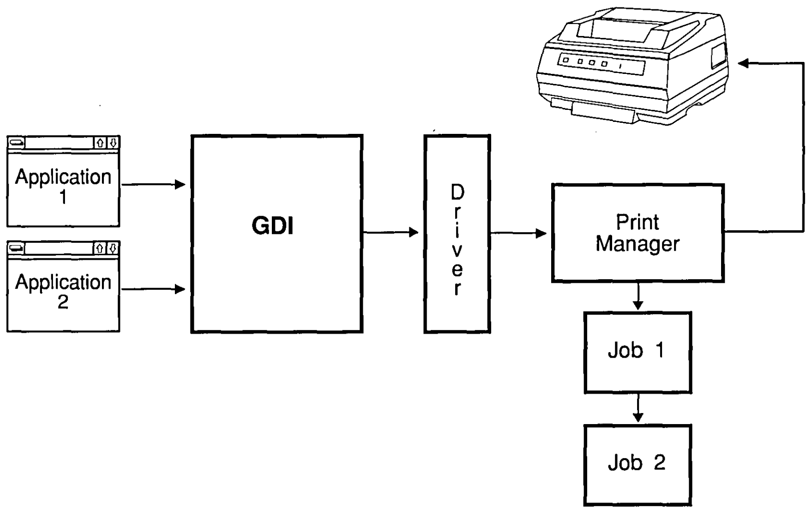


Figure 2.20 Output Path

If only one Windows application were allowed to run at any given time, Print Manager and many of the escape functions would be unnecessary. However, Windows allows several applications to run at once. If two or more of these applications send output simultaneously, each application's output must be separated and remain separated during printing or plotting. Print Manager maintains this separation. The printer-escape functions affect the way Print Manager handles this separation task.

2.16.2 Banding Output

The model used by GDI states that any point on an output device can be written to at any time. This model is easily implemented on vector devices but poses a problem on many dot-matrix devices that cannot scroll backward. Banding provides a solution to this problem.

Banding involves several steps:

1. The application creates a metafile and uses it as an intermediate storage device for the output.

2. Beginning at the top of the metafile, GDI translates a rectangular region (band) of output into device-specific commands, and then sends it to a corresponding job.
3. The application repeats this process until the entire metafile has been converted to bands and the output from these bands has been translated into device-specific commands and stored in a job.
4. The application sends the job to the output device.

When creating a device context, GDI verifies whether the device has banding capabilities. If it does, GDI creates the metafile that will be used during the banding process. To implement banding, you call the necessary output functions and the **NEXTBAND** escape. The **NEXTBAND** escape requires a long pointer to a **RECT** data structure as its output parameter. The device driver copies the coordinates of the next band into this structure. When the entire metafile has been converted into device-specific commands, the driver returns four zeros (0,0,0,0) in the **RECT** structure.

GDI does the banding for you if your output device has banding capabilities and you call the **NEWFRAME** escape. Although **NEWFRAME** requires more memory and is slower, it does simplify the output process. After the application creates each page of output, it calls the **NEWFRAME** escape. If the device is capable of banding, GDI copies output to a metafile and calls the **NEXTBAND** escape for you. As discussed earlier, the **NEXTBAND** escape causes the contents of the metafile to be converted into device-specific commands and to be copied to a corresponding job. If a memory problem occurs or the user terminates a job, the **NEWFRAME** escape returns a message that defines the error or abort message.

2.16.3 Starting and Ending a Print Job

The **STARTDOC** escape informs the device driver that an application is beginning a new print job. After the **STARTDOC** call is issued, Print Manager queues all output from a particular application in a corresponding job until an **ENDDOC** escape is issued. (Note that you cannot use the **ENDDOC** escape to terminate a job.)

2.16.4 Terminating a Print Job

If you send output to a device with the **NEWFRAME** escape, you are required to write a termination procedure and supply it with the application. The **SETABORTPROC** escape sets a pointer to this procedure; it should be called prior to the **STARTDOC** escape. The **ABORTDOC** escape terminates print jobs if it is called before the first call to **NEWFRAME**. It should also be used to terminate jobs that use the **NEXTBAND** escape.

2.16.5 Information Escapes

Four of the escape functions are used to retrieve information about the selected device and its settings. The **GETPHYSPAGESIZE** escape retrieves the physical page size of the output device (in device units), the smallest addressable units on the device. For example, one-fortieth of a millimeter is the smallest addressable unit on some vector devices. A pixel is the smallest addressable unit on a dot-matrix device. The **GETPRINTINGOFFSET** escape retrieves the distance (in device units) from the upper-left corner of the page to the point at which printing begins. The **GETSCALINGFACTOR** escape retrieves the scaling factors for the *x*- and *y*-axes of a device. The scaling factor expresses the number of logical units that are mapped to a device unit. The **QUERYESCSUPPORT** escape determines whether a particular escape function is implemented on a device driver. If the escape in question is implemented, **QUERYESCSUPPORT** returns a nonzero value. If the escape is not implemented, **QUERYESCSUPPORT** returns zero.

2.16.6 Additional Escape Calls

There are two additional escapes that alter the state of the device: the **FLUSHOUTPUT** and **DRAFTMODE** escapes. The **FLUSHOUTPUT** escape flushes the output in the device's buffer (the device stores device operations in the buffer before sending them to Print Manager). The **DRAFTMODE** escape turns on the device's draft mode. This means that the device will use one of its own fonts instead of using a GDI font. It also means that calls to the text-justification functions that alter interword and intercharacter spacing are ignored. For a detailed description of the functions that alter interword and intercharacter spacing, see Sections 2.12, "Text Functions," and 2.13, "Font Functions."

2.17 Environment Functions

Environment functions alter and retrieve information about the environment associated with an output device. The following list briefly describes the two enviornment functions:

Function	Description
GetEnvironment	Copies environment information into a buffer.
SetEnvironment	Copies data to the environment associated with an attached device.

2.18 Summary

Graphics device interface (GDI) functions perform device-independent graphics operations within a Windows application. For more information on topics related to GDI functions, see the following:

Topic	Reference
Function descriptions	*Reference, Volume 1*: Chapter 4, "Functions Directory"
Windows data types and structures	*Reference, Volume 2*: Chapter 7, "Data Types and Structures"
Metafile formats	*Reference, Volume 2*: Chapter 9, "File Formats"
Raster operations	*Reference, Volume 2*: Chapter 11, "Binary and Ternary Raster-Operation Codes"
Printer escapes	*Reference, Volume 2*: Chapter 12, "Printer Escapes"
Drawing text and graphics in a window	*Guide to Programming*: Chapter 3, "Output to a Window"
Drawing bitmaps	*Guide to Programming*: Chapter 11, "Bitmaps"
Sending output to a printer	*Guide to Programming*: Chapter 12, "Printing," and Chapter 17, "Print Settings"
Text fonts	*Guide to Programming*: Chapter 18, "Fonts"
Color palettes	*Guide to Programming*: Chapter 19, "Color Palettes"

Chapter 3 System Services Interface Functions

This chapter describes the system services interface functions. These functions access code and data in modules, allocate and manage both local and global memory, manage tasks, load program resources, translate strings from one character set to another, alter the Microsoft Windows initialization file, assist in system debugging, carry out communications through the system's I/O ports, create and open files, and create sounds using the system's sound generator.

This chapter lists the following categories of functions:

- Module-management functions
- Memory-management functions
- Segment functions
- Operating-system interrupt functions
- Task functions
- Resource-management functions
- String-manipulation functions
- Atom-management functions
- Initialization-file functions
- Communication functions
- Sound functions
- Utility macros and functions
- File I/O functions
- Debugging functions
- Optimization-tool functions
- Application-execution functions

3.1 Module-Management Functions

Module-management functions alter and retrieve information about Windows modules, which are loadable, executable units of code and data. The following list briefly describes each module-management function:

Function	Description
FreeLibrary	Decreases the reference count of a library by one and removes it from memory if the reference count is zero.
FreeModule	Decreases the reference count of a module by one and removes it from memory if the reference count is zero.
FreeProcInstance	Removes a function instance entry at an address.
GetCodeHandle	Determines which code segment contains a specified function.
GetInstanceData	Copies data from an offset in one instance to an offset in another instance.
GetModuleFileName	Copies a module filename.
GetModuleHandle	Returns the module handle of a module.
GetModuleUsage	Returns the reference count of a module.
GetProcAddress	Returns the address of a function in a module.
GetVersion	Returns the current version number of Windows.
LoadLibrary	Loads a library module.
MakeProcInstance	Returns a function-instance address.

3.2 Memory-Management Functions

Memory-management functions manage system memory. There are two categories of functions: those that manage global memory and those that manage local memory. Global memory is all memory in the system that has not been allocated by an application or reserved by the system. Local memory is the memory within a Windows application's data segment. The following list briefly describes each memory-management function:

Function	Description
DefineHandleTable	Creates a private handle table in an application's default data segment.
GetFreeSpace	Retrieves the number of bytes available in the global heap.
GetWinFlags	Retrieves information about the system memory configuration.
GlobalAlloc	Allocates memory from the global heap.
GlobalCompact	Compacts global memory to generate free bytes.
GlobalDiscard	Discards a global memory block if the lock count is zero, but does not invalidate the handle of the memory block.
GlobalDosAlloc	Allocates global memory that can be accessed by DOS running in real or protected mode.
GlobalDosFree	Frees global memory previously allocated by the **GlobalDosAlloc** function.
GlobalFlags	Returns the flags and lock count of a global memory block.
GlobalFree	Removes a global memory block and invalidates the handle of the memory block.
GlobalHandle	Retrieves the handle of a global memory object.
GlobalLock	Retrieves a pointer to a global memory block specified by a handle. Except for nondiscardable objects in protected (standard or 386 enhanced) mode, the block is locked into memory at the given address and its lock count is increased by one.
GlobalLRUNewest	Moves a global memory object to the newest least-recently-used (LRU) position.
GlobalLRUOldest	Moves a global memory object to the oldest least-recently-used (LRU) position.
GlobalNotify	Installs a notification procedure for the current task.
GlobalReAlloc	Reallocates a global memory block.
GlobalSize	Returns the size (in bytes) of a global memory block.

Function	Description
GlobalUnlock	Invalidates the pointer to a global memory block previously retrieved by the **GlobalLock** function. In real mode, or if the block is discardable, **GlobalUnlock** decreases the block's lock count by one.
GlobalUnwire	Decreases the lock count set by the **GlobalWire** function, and unlocks the memory block if the count is zero.
GlobalWire	Moves an object to low memory and increases the lock count.
LimitEMSPages	Limits the amount of expanded memory that Windows will assign to an application.
LocalAlloc	Allocates memory from the local heap.
LocalCompact	Compacts local memory.
LocalDiscard	Discards a local memory block if the lock count is zero, but does not invalidate the handle of the memory block.
LocalFlags	Returns the memory type of a local memory block.
LocalFree	Frees a local memory block from memory if the lock count is zero and invalidates the handle of the memory block.
LocalHandle	Retrieves the handle of a local memory object.
LocalInit	Initializes a local heap in the specified segment.
LocalLock	Locks a block of local memory by increasing its lock count.
LocalReAlloc	Reallocates a local memory block.
LocalShrink	Shrinks the local heap.
LocalSize	Returns the size (in bytes) of a local memory block.
LocalUnlock	Unlocks a local memory block.
LockData	Locks the current data segment in memory.
LockSegment	Locks a specified data segment in memory.
SetSwapAreaSize	Increases the amount of memory that an application reserves for code segments.

Function	Description
SwitchStackBack	Returns the stack of the current task to the task's data segment after it had been previously redirected by the **SwitchTasksBack** function.
SwitchStackTo	Changes the stack of the current task to the specified data segment, such as the data segment of a dynamic-link library (DLL).
UnlockData	Unlocks the current data segment.
UnLockSegment	Unlocks a specified data segment.

3.3 Segment Functions

Segment functions allocate, free, and convert selectors; lock and unlock memory blocks referenced by selectors; and retrieve information about segments. The following list briefly describes each selector function:

Function	Description
AllocDStoCSAlias	Accepts a data-segment selector and returns a code-segment selector that can be used to execute code in a data segment.
AllocSelector	Allocates a new selector.
ChangeSelector	Generates a temporary code selector that corresponds to a given data selector, or a temporary data selector that corresponds to a given code selector.
DefineHandleTable	Creates a private handle table which Windows updates automatically.
FreeSelector	Frees a selector originally allocated by the **AllocSelector**, **AllocCStoDSAlias**, or **AllocDStoCSAlias** functions.
GetCodeInfo	Retrieves information about a code segment.
GlobalFix	Prevents a global memory block from moving in linear memory.

Function	Description
GlobalPageLock	Page-locks the memory associated with the specified global selector and increments its page-lock count. Memory that is page-locked cannot be moved or paged out to disk.
GlobalPageUnlock	Decrements the page-lock count for a block of memory. If the page-lock count reaches zero, the memory can be moved and paged out to disk.
GlobalUnfix	Unlocks a global memory block previously fixed by the **GlobalFix** function.
LockSegment	Locks a segment in memory.
UnlockSegment	Unlocks a segment previously locked by the **LockSegment** function.

NOTE An application should not use these functions unless it is absolutely necessary. Use of these functions violates preferred Windows programming practices.

3.4 Operating-System Interrupt Functions

Operating-system interrupt functions allow an assembly-language application to perform certain DOS and NETBIOS interrupts without directly coding the interrupt. This ensures compatibility with future Microsoft products.

The following list briefly describes these functions:

Function	Description
DOS3Call	Issues a DOS 21H (function-request) interrupt.
NetBIOSCall	Issues a NETBIOS 5CH interrupt.

3.5 Task Functions

Task functions alter the execution status of tasks, return information associated with a task, and retrieve information about the environment in which the task is executing. A task is a single Windows application call. The following list briefly describes each task function:

Function	Description
Catch	Copies the current execution environment to a buffer.
ExitWindows	Initiates the standard Windows shutdown procedure.

Function	Description
GetCurrentPDB	Returns the current DOS Program Data Base (PDB), also known as the Program Segment Prefix (PSP).
GetCurrentTask	Returns the task handle of the current task.
GetDOSEnvironment	Retrieves the environment string of the currently running task.
GetNumTasks	Returns the number of tasks currently executing in the system.
SetErrorMode	Controls whether Windows handles DOS Function 24H errors or allows the calling application to handle them.
Throw	Restores the execution environment to the specified values.
Yield	Halts the current task and starts any waiting task.

3.6 Resource-Management Functions

Resource-management functions find and load application resources from a Windows executable file. A resource can be a cursor, icon, bitmap, string, or font. The following list briefly describes each resource-management function:

Function	Description
AccessResource	Opens the specified resource.
AllocResource	Allocates uninitialized memory for a resource.
FindResource	Determines the location of a resource.
FreeResource	Removes a loaded resource from memory.
LoadAccelerators	Loads an accelerator table.
LoadBitmap	Loads a bitmap resource.
LoadCursor	Loads a cursor resource.
LoadIcon	Loads an icon resource.
LoadMenu	Loads a menu resource.
LoadResource	Loads a resource.
LoadString	Loads a string resource.
LockResource	Retrieves the absolute memory address of a resource.

Function	Description
SetResourceHandler	Sets up a function to load resources.
SizeofResource	Supplies the size (in bytes) of a resource.
UnlockResource	Unlocks a resource.

3.7 String-Manipulation Functions

String-manipulation functions translate strings from one character set to another, determine and convert the case of strings, determine whether a character is alphabetic or alphanumeric, find adjacent characters in a string, and perform other string manipulation. The following list briefly describes each string-translation function:

Function	Description
AnsiLower	Converts a character string to lowercase.
AnsiLowerBuff	Converts a character string in a buffer to lowercase.
AnsiNext	Returns a long pointer to the next character in a string.
AnsiPrev	Returns a long pointer to the previous character in a string.
AnsiToOem	Converts an ANSI string to an OEM character string.
AnsiToOemBuff	Converts an ANSI string in a buffer to an OEM character string.
AnsiUpper	Converts a character string to uppercase.
AnsiUpperBuff	Converts a character string in a buffer to uppercase.
IsCharAlpha	Determines whether a character is alphabetical.
IsCharAlphaNumeric	Determines whether a character is alphanumeric.
IsCharLower	Determines whether a character is lowercase.
IsCharUpper	Determines whether a character is uppercase.
lstrcat	Concatenates two strings identified by long pointers.
lstrcmp	Performs a case-sensitive comparison of two strings identified by long pointers.
lstrcmpi	Performs a case-insensitive comparison of two strings identified by long pointers.

Function	Description
lstrcpy	Copies one string to another; both strings are identified by long pointers.
lstrlen	Determines the length of a string identified by a long pointer.
OemToAnsi	Converts an OEM character string to an ANSI string.
OemToAnsiBuff	Converts an OEM character string in a buffer to an ANSI string.
ToAscii	Translates a virtual-key code to the corresponding ANSI character or characters.
wsprintf	Formats and stores a series of characters and values in a buffer. Format arguments are passed separately.
wvsprintf	Formats and stores a series of characters and values in a buffer. Format arguments are passed through an array.

3.8 Atom-Management Functions

Atom-management functions create and manipulate atoms. Atoms are integers that uniquely identify character strings. They are useful in applications that use many character strings and in applications that need to conserve memory. Windows stores atoms in atom tables. A local atom table is allocated in an application's data segment; it cannot be accessed by other applications. The global atom table can be shared, and is useful in applications that use dynamic data exchange (DDE). The following list briefly describes each atom-management function:

Function	Description
AddAtom	Creates an atom for a character string.
DeleteAtom	Deletes an atom if the reference count is zero.
FindAtom	Retrieves an atom associated with a character string.
GetAtomHandle	Retrieves a handle (relative to the local heap) of the string that corresponds to a specified atom.
GetAtomName	Copies the character string associated with an atom.
GlobalAddAtom	Creates a global atom for a character string.
GlobalDeleteAtom	Deletes a global atom if the reference count is zero.

Function	Description
GlobalFindAtom	Retrieves a global atom associated with a character string.
GlobalGetAtomName	Copies the character string associated with a global atom.
InitAtomTable	Initializes an atom hash table.
MAKEINTATOM	Casts an integer for use as a function argument.

3.9 Initialization-File Functions

Initialization-file functions obtain information from and copy information to the Windows initialization file WIN.INI and private initialization files. A Windows initialization file is a special ASCII file that contains key-name–value pairs that represent run-time options for applications. The following list briefly describes each initialization-file function:

Function	Description
GetPrivateProfileInt	Returns an integer value in a section from a private initialization file.
GetPrivateProfileString	Returns a character string in a section from a private initialization file.
GetProfileInt	Returns an integer value in a section from the WIN.INI file.
GetProfileString	Returns a character string in a section from the WIN.INI file.
WritePrivateProfileString	Copies a character string to a private initialization file, or deletes one or more lines in a private initialization file.
WriteProfileString	Copies a character string to the WIN.INI file, or deletes one or more lines from WIN.INI.

An application should use a private (application-specific) initialization file to record information which affects only that application. This improves both the performance of the application and Windows itself by reducing the amount of information that Windows must read when it accesses the initialization file. An application should record information in WIN.INI only if it affects the Windows environment or other applications; in such cases, the application should send the WM_WININICHANGE message to all top-level windows.

The files WININI.TXT and SYSINI.TXT supplied with the retail version of Windows describe the contents of WIN.INI and SYSTEM.INI, respectively.

3.10 Communication Functions

Communication functions carry out communications through the system's serial and parallel I/O ports. The following list briefly describes each communication function:

Function	Description
BuildCommDCB	Fills a device control block with control codes.
ClearCommBreak	Clears the communication break state from a communication device.
CloseComm	Closes a communication device after transmitting the current buffer.
EscapeCommFunction	Directs a device to carry out an extended function.
FlushComm	Flushes characters from a communication device.
GetCommError	Fills a buffer with the communication status.
GetCommEventMask	Retrieves, then clears, an event mask.
GetCommState	Fills a buffer with a device control block.
OpenComm	Opens a communication device.
ReadComm	Reads the bytes from a communication device into a buffer.
SetCommBreak	Sets a break state on the communication device.
SetCommEventMask	Retrieves and then sets an event mask on the communication device.
SetCommState	Sets a communication device to the state specified by the device control block.
TransmitCommChar	Places a character at the head of the transmit queue.
UngetCommChar	Specifies which character will be the next character to be read.
WriteComm	Writes the bytes from a buffer to a communication device.

3.11 Sound Functions

Sound functions create sound and music for the system's sound generator. The following list briefly describes each sound function:

Function	Description
CloseSound	Closes the play device after flushing the voice queues and freeing the buffers.
CountVoiceNotes	Returns the number of notes in the specified queue.
GetThresholdEvent	Returns a long pointer to a threshold flag.
GetThresholdStatus	Returns the threshold-event status for each voice.
OpenSound	Opens the play device for exclusive use.
SetSoundNoise	Sets the source and duration of a noise from the play device.
SetVoiceAccent	Places an accent in the voice queue.
SetVoiceEnvelope	Places the voice envelope in the voice queue.
SetVoiceNote	Places a note in the specified voice queue.
SetVoiceQueueSize	Allocates a specified number of bytes for the voice queue.
SetVoiceSound	Places the specified sound frequency and durations in a voice queue.
SetVoiceThreshold	Sets the threshold level for a given voice.
StartSound	Starts playing each voice queue.
StopSound	Stops playing all voice queues and flushes their contents.
SyncAllVoices	Places a sync mark in each voice queue.
WaitSoundState	Waits until the play driver enters the specified state.

3.12 Utility Macros and Functions

Utility macros and functions return contents of words and bytes, create unsigned long integers and data structures, and perform specialized arithmetic. The following list briefly describes each utility macro or function:

Function	Description
HIBYTE	Returns the high-order byte of an integer.
HIWORD	Returns the high-order word of a long integer.
LOBYTE	Returns the low-order byte of an integer.
LOWORD	Returns the low-order word of a long integer.
MAKEINTATOM	Casts an integer for use as a function argument.
MAKEINTRESOURCE	Converts an integer value into a long pointer to a string, with the high-order word of the long pointer set to zero.
MAKELONG	Creates an unsigned long integer.
MAKEPOINT	Converts a long value that contains the *x*- and *y*-coordinates of a point into a **POINT** data structure.
MulDiv	Multiplies two word-length values and then divides the result by a third word-length value, returning the result rounded to the nearest integer.
PALETTEINDEX	Converts an integer into a palette-index **COLORREF** value.
PALETTERGB	Converts three values for red, green, and blue into a palette-relative RGB **COLORREF** value.
RGB	Converts three values for red, green, and blue into an explicit RGB **COLORREF** value.

3.13 File I/O Functions

File I/O functions create, open, read from, write to, and close files. The following list briefly describes each file I/O function:

Function	Description
GetDriveType	Determines whether a disk drive is removeable, fixed, or remote.
GetSystemDirectory	Retrieves the pathname of the Windows system subdirectory.
GetTempDrive	Returns the letter of the optimal drive for temporary file storage.
GetTempFileName	Creates a temporary filename.
GetWindowsDirectory	Retrieves the pathname of the Windows directory.

Function	Description
_lclose	Closes a file.
_lcreat	Creates a new file or opens and truncates an existing file.
_llseek	Positions the pointer to a file.
_lopen	Opens an existing file.
_lread	Reads data from a file.
_lwrite	Writes data in a file.
OpenFile	Creates, opens, reopens, or deletes the specified file.
SetHandleCount	Changes the number of file handles available to a task.

3.14 Debugging Functions

Debugging functions help locate programming errors in an application or library. The following briefly describes these functions:

Function	Description
DebugBreak	Forces a break to the debugger.
FatalAppExit	Displays a message box and then terminates the application.
FatalExit	Displays the current state of Windows and prompts for instructions on how to proceed.
OutputDebugString	Sends a debugging message to the debugger if present, or to the AUX device if the debugger is not present.
ValidateCodeSegments	Determines whether any code segments have been altered by random memory overwrites.
ValidateFreeSpaces	Checks free segments in memory for valid contents.

3.15 Optimization-Tool Functions

Optimization-tool functions control how the Windows Profiler and Swap software development tools interact with an application being developed. The following list briefly describes these functions:

Function	Description
ProfClear	Discards all samples in the Profiler sampling buffer.
ProfFinish	Stops sampling by Profiler and flushes the buffer to disk.
ProfFlush	Flushes the Profiler sampling buffer to disk.
ProfInsChk	Determines if Profiler is installed.
ProfSampRate	Sets the rate of code sampling by Profiler.
ProfSetup	Sets up the Profiler sampling buffer and recording rate.
ProfStart	Starts sampling by Profiler.
ProfStop	Stops sampling by Profiler.
SwapRecording	Begins or ends analyzing by Swap of the application's swapping behavior.

3.16 Application-Execution Functions

Application-execution tasks permit one application to execute another program. The following list briefly describe these functions:

Function	Description
LoadModule	Executes a separate application.
WinExec	Executes a separate application.
WinHelp	Runs the Windows Help application and passes context or topic information to Help.

The **WinExec** function provides a high-level method for executing any Windows or standard DOS application. The calling application supplies a string containing the name of the executable file to be run and any command parameters, and specifies the initial state of the application's window.

The **LoadModule** function is similar, but provides more control over the environment in which the application is executed. The calling application supplies the name of the executable file and a DOS Function 4BH, Code 00H, parameter block.

The **WinHelp** function executes the Windows Help application and optionally passes data to it indicating the nature of the help requested by the application. This data is either an integer which specifies a context identifier in the help file or a string containing a key word in the help file.

3.17 Summary

System services interface functions access code and data in modules, allocate and manage both local and global memory, manage tasks, load program resources, translate strings from one character set to another, alter the Windows initialization file, assist in system debugging, carry out communications through the system's I/O ports, create and open files, and create sounds using the system's sound generator. For more information on topics related to system services interface functions, see the following:

Topic	Reference
Function descriptions	*Reference, Volume 1:* Chapter 4, "Functions Directory"
Windows data types and structures	*Reference, Volume 2:* Chapter 7, "Data Types and Structures"
Initialization-file formats	*Reference, Volume 2:* Chapter 9, "File Formats"
Diagnostic messages for debugging	*Reference, Volume 2:* Appendix C, "Windows Debugging Messages"
Writing and reading from files	*Guide to Programming:* Chapter 10, "File Input and Output"
Managing memory	*Guide to Programming:* Chapter 15, "Memory Management," and Chapter 16, "More Memory Management"
Libraries	*Guide to Programming:* Chapter 20, "Dynamic-Link Libraries"
Using Profiler	*Tools:* Chapter 13, "Analyzing CPU Time: Profiler"
Using Swap	*Tools:* Chapter 14, "Analyzing Swaps: Swap"

Chapter 4

Functions Directory

This chapter contains an alphabetical list of functions from the Microsoft Windows application programming interface (API). The documentation for each function contains a line illustrating correct syntax, a statement about the function's purpose, a description of its input parameters, and a description of its return value. The documentation for some functions contains additional, important information that an application developer needs in order to use the function.

AccessResource

Syntax **int** AccessResource(*hInstance*, *hResInfo*)

This function opens the specified resource file and moves the file pointer to the beginning of the specified resource, letting an application read the resource from the file. The **AccessResource** function supplies a DOS file handle that can be used in subsequent file-read calls to load the resource. The file is opened for reading only.

Applications that use this function must close the resource file by calling the **_lclose** function after reading the resource.

Parameter	Type/Description
hInstance	**HANDLE** Identifies the instance of the module whose executable file contains the resource.
hResInfo	**HANDLE** Identifies the desired resource. This handle should be created by using the **FindResource** function.

Return Value The return value specifies a DOS file handle to the designated resource file. It is –1 if the resource cannot be found.

Comments **AccessResource** can exhaust available DOS file handles and cause errors if the opened file is not closed after the resource is accessed.

AddAtom

Syntax ATOM **AddAtom**(*lpString*)

This function adds the character string pointed to by the *lpString* parameter to the atom table and creates a new atom that uniquely identifies the string. The atom can be used in a subsequent **GetAtomName** function to retrieve the string from the atom table.

The **AddAtom** function stores no more than one copy of a given string in the atom table. If the string is already in the table, the function returns the existing atom value and increases the string's reference count by one.

Parameter	Type/Description
lpString	**LPSTR** Points to the character string to be added to the table. The string must be a null-terminated character string.

Return Value The return value specifies the newly created atom if the function is successful. Otherwise, it is NULL.

Comments The atom values returned by **AddAtom** range from 0xC000 to 0xFFFF. Atoms are case insensitive.

AddFontResource

Syntax **int AddFontResource**(*lpFilename*)

This function adds the font resource from the file named by the *lpFilename* parameter to the Windows font table. The font can subsequently be used by any application.

Parameter	Type/Description
lpFilename	**LPSTR** Points to a character string that names the font-resource file or contains a handle to a loaded module. If *lpFilename* points to the font-resource filename, the string must be null-terminated, have the DOS filename format, and include the extension. If *lpFilename* contains a handle, the handle is in the low-order word and the high-order word is zero.

Return Value The return value specifies the number of fonts added. The return value is zero if no fonts are loaded.

Comments Any application that adds or removes fonts from the Windows font table should notify other windows of the change by using the **SendMessage** function with the *hWnd* parameter set to –1 to send a WM_FONTCHANGE message to all top-level windows in the system.

It is good practice to remove any font resource an application has added once the application is through with the resource.

For a description of font resources, see the *Guide to Programming*.

AdjustWindowRect

Syntax **void AdjustWindowRect**(*lpRect*, *dwStyle*, *bMenu*)

This function computes the required size of the window rectangle based on the desired client-rectangle size. The window rectangle can then be passed to the **CreateWindow** function to create a window whose client area is the desired size. A client rectangle is the smallest rectangle that completely encloses a client area. A window rectangle is the smallest rectangle that completely encloses the window. The dimensions of the resulting window rectangle depend on the window styles and on whether the window has a menu.

Parameter	Type/Description
lpRect	**LPRECT** Points to a **RECT** data structure that contains the coordinates of the client rectangle.
dwStyle	**DWORD** Specifies the window styles of the window whose client rectangle is to be converted.
bMenu	**BOOL** Specifies whether the window has a menu.

Return Value None.

Comments This function assumes a single menu row. If the menu bar wraps to two or more rows, the coordinates are incorrect.

AdjustWindowRectEx 3.0

Syntax **void AdjustWindowRectEx**(*lpRect*, *dwStyle*, *bMenu*, *dwExStyle*)

This function computes the required size of the rectangle of a window with extended style based on the desired client-rectangle size. The window rectangle can then be passed to the **CreateWindowEx** function to create a window whose client area is the desired size.

A client rectangle is the smallest rectangle that completely encloses a client area. A window rectangle is the smallest rectangle that completely encloses the window. The dimensions of the resulting window rectangle depends on the window styles and on whether the window has a menu.

Parameter	Type/Description
lpRect	**LPRECT** Points to a **RECT** data structure that contains the coordinates of the client rectangle.
dwStyle	**DWORD** Specifies the window styles of the window whose client rectangle is to be converted.
bMenu	**BOOL** Specifies whether the window has a menu.
dwExStyle	**DWORD** Specifies the extended style of the window being created.

Return Value None.

Comments This function assumes a single menu row. If the menu bar wraps to two or more rows, the coordinates are incorrect.

AllocDStoCSAlias 3.0

Syntax **WORD AllocDStoCSAlias**(*wSelector*)

This function accepts a data-segment selector and returns a code-segment selector that can be used to execute code in the data segment. When in protected mode, attempting to execute code directly in a data segment will cause a general protection violation. **AllocDStoCSAlias** allows an application to execute code which the application had created in its own stack segment.

The application must free the new selector by calling the **FreeSelector** function.

Parameter	Type/Description
wSelector	**WORD** Specifies the data-segment selector.

Return Value The return value is the code-segment selector corresponding to the data-segment selector. If the function cannot allocate a new selector, the return value is zero.

Comments Windows does not track segment movements. Consequently, the data segment must be fixed and nondiscardable; otherwise, the data segment might move, invalidating the code-segment selector.

The **ChangeSelector** function provides another method of obtaining a code selector corresponding to a data selector.

An application should not use this function unless it is absolutely necessary. Use of this function violates preferred Windows programming practices.

AllocResource

Syntax **HANDLE AllocResource**(*hInstance*, *hResInfo*, *dwSize*)

This function allocates uninitialized memory for the passed resource. All resources must be initially allocated by using the **AllocResource** function. The **LoadResource** function calls this function before loading the resource.

Parameter	Type/Description
hInstance	**HANDLE** Identifies the instance of the module whose executable file contains the resource.
hResInfo	**HANDLE** Identifies the desired resource. It is assumed that this handle was created by using the **FindResource** function.
dwSize	**DWORD** Specifies an override size in bytes to allocate for the resource. The override is ignored if the size is zero.

Return Value The return value identifies the global memory block allocated for the resource.

AllocSelector 3.0

Syntax **WORD AllocSelector**(*wSelector*)

This function allocates a new selector. If the *wSelector* parameter is a valid selector, **AllocSelector** returns a new selector which is an exact copy of the one specified by *wSelector*. If *wSelector* is NULL, **AllocSelector** returns a new, uninitialized selector.

The application must free the new selector by calling the **FreeSelector** function.

Parameter	Type/Description
wSelector	**WORD** Specifies the selector to be copied, or NULL if **AllocSelector** is to allocate a new, uninitialized selector.

Return Value The return value is either a selector that is a copy of an existing selector, or a new, uninitialized selector. If the function could not allocate a new selector, the return value is zero.

Comments An application can call **AllocSelector** to allocate a selector that it can pass to the **ChangeSelector** function.

An application should not use this function unless it is absolutely necessary. Use of this function violates preferred Windows programming practices.

AnimatePalette 3.0

Syntax void **AnimatePalette**(*hPalette*, *wStartIndex*, *wNumEntries*, *lpPaletteColors*)

This function replaces entries in the logical palette identified by the *hPalette* parameter. When an application calls **AnimatePalette**, it does not have to update its client area because Windows maps the new entries into the system palette immediately.

Parameter	Type/Description
hPalette	**HPALETTE** Identifies the logical palette.
wStartIndex	**WORD** Specifies the first entry in the palette to be animated.
wNumEntries	**WORD** Specifies the number of entries in the palette to be animated.
lpPaletteColors	**LPPALETTEENTRY** Points to the first member of an array of **PALETTEENTRY** data structures to replace the palette entries identified by *wStartIndex* and *wNumEntries*.

Return Value None.

Comments **AnimatePalette** will only change entries with the PC_RESERVED flag set in the corresponding **palPaletteEntry** field of the **LOGPALETTE** data structure that defines the current logical palette. The **CreatePalette** function creates a logical palette.

AnsiLower

Syntax LPSTR **AnsiLower**(*lpString*)

This function converts the given character string to lowercase. The conversion is made by the language driver based on the criteria of the current language selected by the user at setup or with the Control Panel.

Parameter	Type/Description
lpString	**LPSTR** Points to a null-terminated character string or specifies single character. If *lpString* specifies single character, that character is in the low-order byte of the low-order word, and the high-order word is zero.

Return Value The return value points to a converted character string if the function parameter is a character string. Otherwise, it is a 32-bit value that contains the converted character in the low-order byte of the low-order word.

AnsiLowerBuff 3.0

Syntax **WORD AnsiLowerBuff**(*lpString*, *nLength*)

This function converts character string in a buffer to lowercase. The conversion is made by the language driver based on the criteria of the current language selected by the user at setup or with the Control Panel.

Parameter	Type/Description
lpString	**LPSTR** Points to a buffer containing one or more characters.
nLength	**WORD** Specifies the number of characters in the buffer identified by the *lpString* parameter. If *nLength* is zero, the length is 64K (65,536).

Return Value The return value specifies the length of the converted string.

AnsiNext

Syntax **LPSTR AnsiNext**(*lpCurrentChar*)

This function moves to the next character in a string.

Parameter	Type/Description
lpCurrentChar	**LPSTR** Points to a character in a null-terminated string.

Return Value The return value points to the next character in the string, or, if there is no next character, to the null character at the end of the string.

Comments The **AnsiNext** function is used to move through strings whose characters are two or more bytes each (for example, strings that contain characters from a Japanese character set).

AnsiPrev

Syntax

LPSTR AnsiPrev(*lpStart*, *lpCurrentChar*)

This function moves to the previous character in a string.

Parameter	Type/Description
lpStart	**LPSTR** Points to the beginning of the string.
lpCurrentChar	**LPSTR** Points to a character in a null-terminated string.

Return Value

The return value points to the previous character in the string, or to the first character in the string if the *lpCurrentChar* parameter is equal to the *lpStart* parameter.

Comments

The **AnsiPrev** function is used to move through strings whose characters are two or more bytes each (for example, strings that contain characters from a Japanese character set).

AnsiToOem

Syntax

int AnsiToOem(*lpAnsiStr*, *lpOemStr*)

This function translates the string pointed to by the *lpAnsiStr* parameter from the ANSI character set into the OEM-defined character set. The string can be greater than 64K in length.

Parameter	Type/Description
lpAnsiStr	**LPSTR** Points to a null-terminated string of characters from the ANSI character set.
lpOemStr	**LPSTR** Points to the location where the translated string is to be copied. The *lpOemStr* parameter can be the same as *lpAnsiStr* to translate the string in place.

Return Value

The return value is always –1.

AnsiToOemBuff 3.0

Syntax void **AnsiToOemBuff**(*lpAnsiStr*, *lpOemStr*, *nLength*)

This function translates the string in the buffer pointed to by the *lpAnsiStr* parameter from the ANSI character set into the OEM-defined character set.

Parameter	Type/Description
lpAnsiStr	**LPSTR** Points to a buffer containing one or more characters from the ANSI character set.
lpOemStr	**LPSTR** Points to the location where the translated string is to be copied. The *lpOemStr* parameter can be the same as *lpAnsiStr* to translate the string in place.
nLength	**WORD** Specifies the number of characters in the buffer identified by the *lpAnsiStr* parameter. If *nLength* is zero, the length is 64K (65,536).

Return Value None.

AnsiUpper

Syntax LPSTR **AnsiUpper**(*lpString*)

This function converts the given character string to uppercase. The conversion is made by the language driver based on the criteria of the current language selected by the user at setup or with the Control Panel.

Parameter	Type/Description
lpString	**LPSTR** Points to a null-terminated character string or specifies single character. If *lpString* specifies a single character, that character is in the low-order byte of the low-order word, and the high-order word is zero.

Return Value The return value points to a converted character string if the function parameter is a character string; otherwise, it is a 32-bit value that contains the converted character in the low-order byte of the low-order word.

AnsiUpperBuff 3.0

Syntax **WORD** **AnsiUpperBuff**(*lpString*, *nLength*)

This function converts a character string in a buffer to uppercase. The conversion is made by the language driver based on the criteria of the current language selected by the user at setup or with the Control Panel.

Parameter	Type/Description
lpString	**LPSTR** Points to a buffer containing one or more characters.
nLength	**WORD** Specifies the number of characters in the buffer identified by the *lpString* parameter. If *nLength* is zero, the length is 64K (65,536).

Return Value The return value specifies the length of the converted string.

AnyPopup

Syntax **BOOL** **AnyPopup**( )

This function indicates whether a pop-up window exists on the screen. It searches the entire Windows screen, not just the caller's client area. The **AnyPopup** function returns nonzero even if a pop-up window is completely covered by another window.

This function has no parameters.

Return Value The return value is nonzero if a pop-up window exists. Otherwise, it is zero.

AppendMenu 3.0

Syntax **BOOL** **AppendMenu**(*hMenu*, *wFlags*, *wIDNewItem*, *lpNewItem*)

This function appends a new item to the end of a menu. The application can specify the state of the menu item by setting values in the *wFlags* parameter.

Parameter	Type/Description
hMenu	**HMENU** Identifies the menu to be changed.
wFlags	**WORD** Specifies information about the state of the new menu item when it is added to the menu. It consists of one or more values listed in the following "Comments" section.
wIDNewItem	**WORD** Specifies either the command ID of the new menu item or, if *wFlags* is set to MF_POPUP, the menu handle of the pop-up menu.
lpNewItem	**LPSTR** Specifies the content of the new menu item. The interpretation of the *lpNewItem* parameter depends upon the setting of the *wFlags* parameter.

If *wFlags* is	*lpNewItem*
MF_STRING	Contains a long pointer to a null-terminated character string.
MF_BITMAP	Contains a bitmap handle **HBITMAP** in its low-order word.
MF_OWNERDRAW	Contains an application-supplied 32-bit value which the application can use to maintain additional data associated with the menu item. This 32-bit value is available to the application in the **itemData** field of the structure pointed to by the *lParam* parameter of the WM_MEASUREITEM and WM_DRAWITEM messages sent when the menu item is initially displayed or is changed.

Return Value

The return value specifies the outcome of the function. It is TRUE if the function is successful. Otherwise, it is FALSE.

Comments

Whenever a menu changes (whether or not the menu resides in a window that is displayed), the application should call **DrawMenuBar**.

Each of the following groups lists flags that are mutually exclusive and should not be used together:

- MF_BYCOMMAND and MF_BYPOSITION
- MF_DISABLED, MF_ENABLED, and MF_GRAYED
- MF_BITMAP, MF_STRING, and MF_OWNERDRAW
- MF_MENUBARBREAK and MF_MENUBREAK
- MF_CHECKED and MF_UNCHECKED

The following list describes the flags which may be set in the *wFlags* parameter:

Value	Meaning
MF_BITMAP	Uses a bitmap as the item. The low-order word of the *lpNewItem* parameter contains the handle of the bitmap.
MF_CHECKED	Places a checkmark next to the item. If the application has supplied checkmark bitmaps (see **SetMenuItemBitmaps**), setting this flag displays the "checkmark on" bitmap next to the menu item.
MF_DISABLED	Disables the menu item so that it cannot be selected, but does not gray it.
MF_ENABLED	Enables the menu item so that it can be selected and restores it from its grayed state.
MF_GRAYED	Disables the menu item so that it cannot be selected and grays it.
MF_MENUBARBREAK	Same as MF_MENUBREAK except that for pop-up menus, separates the new column from the old column with a vertical line.
MF_MENUBREAK	Places the item on a new line for static menu-bar items. For pop-up menus, places the item in a new column, with no dividing line between the columns.
MF_OWNERDRAW	Specifies that the item is an owner-draw item. The window that owns the menu receives a WM_MEASUREITEM message when the menu is displayed for the first time to retrieve the height and width of the menu item. The WM_DRAWITEM message is then sent whenever the owner must update the visual appearance of the menu item. This option is not valid for a top-level menu item.

Value	Meaning
MF_POPUP	Specifies that the menu item has a pop-up menu associated with it. The *wIDNewItem* parameter specifies a handle to a pop-up menu to be associated with the item. This is used for adding either a top-level pop-up menu or adding a hierarchical pop-up menu to a pop-up menu item.
MF_SEPARATOR	Draws a horizontal dividing line. Can only be used in a pop-up menu. This line cannot be grayed, disabled, or highlighted. The *lpNewItem* and *wIDNewItem* parameters are ignored.
MF_STRING	Specifies that the menu item is a character string; the *lpNewItem* parameter points to the string for the menu item.
MF_UNCHECKED	Does not place a checkmark next to the item (default). If the application has supplied checkmark bitmaps (see **SetMenuItemBitmaps**), setting this flag displays the "checkmark off" bitmap next to the menu item.

Arc

Syntax **BOOL** **Arc**(*hDC*, *X1*, *Y1*, *X2*, *Y2*, *X3*, *Y3*, *X4*, *Y4*)

This function draws an elliptical arc. The center of the arc is the center of the bounding rectangle specified by the points *(X1, Y1)* and *(X2, Y2)*. The arc starts at the point *(X3, Y3)* and ends at the point *(X4, Y4)*. The arc is drawn using the selected pen and moving in a counterclockwise direction. Since an arc does not define a closed area, it is not filled.

Parameter	Type/Description
hDC	**HDC** Identifies the device context.
X1	**int** Specifies the logical *x*-coordinate of the upper-left corner of the bounding rectangle.
Y1	**int** Specifies the logical *y*-coordinate of the upper-left corner of the bounding rectangle.
X2	**int** Specifies the logical *x*-coordinate of the lower-right corner of the bounding rectangle.

Parameter	Type/Description
Y2	**int** Specifies the logical *y*-coordinate of the lower-right corner of the bounding rectangle.
X3	**int** Specifies the logical *x*-coordinate of the arc's starting point. This point does not have to lie exactly on the arc.
Y3	**int** Specifies the logical *y*-coordinate of the arc's starting point. This point does not have to lie exactly on the arc.
X4	**int** Specifies the logical *x*-coordinate of the arc's endpoint. This point does not have to lie exactly on the arc.
Y4	**int** Specifies the logical *y*-coordinate of the arc's endpoint. This point does not have to lie exactly on the arc.

Return Value The return value specifies whether the arc is drawn. It is nonzero if the arc is drawn. Otherwise, it is zero.

Comments The width of the rectangle specified by the absolute value of $X2 - X1$ must not exceed 32,767 units. This limit applies to the height of the rectangle as well.

ArrangeIconicWindows 3.0

Syntax **WORD ArrangeIconicWindows**(*hWnd*)

This function arranges all the minimized (iconic) child windows of the window specified by the *hWnd* parameter.

Parameter	Type/Description
hWnd	**HWND** Identifies the window.

Return Value The return value is the height of one row of icons, or zero if there were no icons.

Comments Applications that maintain their own iconic child windows call this function to arrange icons in a client window. This function also arranges icons on the desktop window, which covers the entire screen. The **GetDesktopWindow** function retrieves the window handle of the desktop window.

To arrange iconic MDI child windows in an MDI client window, an application sends the WM_MDIICONARRANGE message to the MDI client window.

BeginDeferWindowPos 3.0

Syntax HANDLE **BeginDeferWindowPos**(*nNumWindows*)

This function allocates memory to contain a multiple window-position data structure and returns a handle to the structure. The **DeferWindowPos** function fills this data structure with information about the target position for a window that is about to be moved. The **EndDeferWindowPos** function accepts this data structure and instantaneously repositions the windows using the information stored in the structure.

Parameter	Type/Description
nNumWindows	**int** Specifies the initial number of windows for which position information is to be stored in the data structure. The **DeferWindowPos** function increases the size of the structure if needed.

Return Value The return value identifies the multiple window-position data structure. The return value is NULL if system resources are not available to allocate the structure.

BeginPaint

Syntax HDC **BeginPaint**(*hWnd*, *lpPaint*)

This function prepares the given window for painting and fills the paint structure pointed to by the *lpPaint* parameter with information about the painting.

The paint structure contains a handle to the device context for the window, a **RECT** data structure that contains the smallest rectangle that completely encloses the update region, and a flag that specifies whether or not the background has been erased.

The **BeginPaint** function automatically sets the clipping region of the device context to exclude any area outside the update region. The update region is set by the **InvalidateRect** or **InvalidateRgn** functions and by the system after sizing, moving, creating, scrolling, or any other operation that affects the client area. If the update region is marked for erasing, **BeginPaint** sends a WM_ERASEBKGND message to the window.

An application should not call the **BeginPaint** function except in response to a WM_PAINT message. Each **BeginPaint** call must have a matching call to the **EndPaint** function.

Parameter	Type/Description
hWnd	**HWND** Identifies the window to be repainted.
lpPaint	**LPPAINTSTRUCT** Points to the **PAINTSTRUCT** data structure that is to receive painting information, such as the device context for the window and the update rectangle.

Return Value The return value identifies the device context for the specified window.

Comments If the caret is in the area to be painted, the **BeginPaint** function automatically hides the caret to prevent it from being erased.

BitBlt

Syntax **BOOL BitBlt**(*hDestDC*, *X*, *Y*, *nWidth*, *nHeight*, *hSrcDC*, *XSrc*, *YSrc*, *dwRop*)

This function moves a bitmap from the source device given by the *hSrcDC* parameter to the destination device given by the *hDestDC* parameter. The *XSrc* and *YSrc* parameters specify the origin on the source device of the bitmap that is to be moved. The *X*, *Y*, *nWidth*, and *nHeight* parameters specify the origin, width, and height of the rectangle on the destination device that is to be filled by the bitmap. The *dwRop* parameter (raster operation) defines how the bits of the source and destination are combined.

Parameter	Type/Description
hDestDC	**HDC** Identifies the device context that is to receive the bitmap.
X	**int** Specifies the logical *x*-coordinate of the upper-left corner of the destination rectangle.
Y	**int** Specifies the logical *y*-coordinate of the upper-left corner of the destination rectangle.
nWidth	**int** Specifies the width (in logical units) of the destination rectangle and source bitmap.
nHeight	**int** Specifies the height (in logical units) of the destination rectangle and source bitmap.
hSrcDC	**HDC** Identifies the device context from which the bitmap will be copied. It must be NULL if the *dwRop* parameter specifies a raster operation that does not include a source.

Parameter	Type/Description
XSrc	**int** Specifies the logical *x*-coordinate of the upper-left corner of the source bitmap.
YSrc	**int** Specifies the logical *y*-coordinate of the upper-left corner of the source bitmap.
dwRop	**DWORD** Specifies the raster operation to be performed. Raster-operation codes define how the graphics device interface (GDI) combines colors in output operations that involve a current brush, a possible source bitmap, and a destination bitmap. For a list of raster-operation codes, see Table 4.1, "Raster Operations."

Return Value

The return value specifies whether the bitmap is drawn. It is nonzero if the bitmap is drawn. Otherwise, it is zero.

Comments

GDI transforms the *nWidth* and *nHeight* parameters, once by using the destination display context, and once by using the source display context. If the resulting extents do not match, GDI uses the **StretchBlt** function to compress or stretch the source bitmap as necessary. If destination, source, and pattern bitmaps do not have the same color format, the **BitBlt** function converts the source and pattern bitmaps to match the destination. The foreground and background colors of the destination are used in the conversion.

If **BitBlt** converts monochrome bitmaps to color, it sets white bits (1) to the background color and black bits (0) to the foreground color. The foreground and background colors of the destination device context are used. To convert color to monochrome, **BitBlt** sets pixels that match the background color to white (1), and sets all other pixels to black (0). The foreground and background colors of the color-source device context are used.

The foreground color is the current text color for the specified device context, and the background color is the current background color for the specified device context.

Not all devices support the **BitBlt** function. For more information, see the RC_BITBLT raster capability in the **GetDeviceCaps** function, later in this chapter.

Table 4.1 lists the various raster-operation codes for the *dwRop* parameter:

Table 4.1 Raster Operations

Code	Description
BLACKNESS	Turns all output black.
DSTINVERT	Inverts the destination bitmap.
MERGECOPY	Combines the pattern and the source bitmap using the Boolean AND operator.
MERGEPAINT	Combines the inverted source bitmap with the destination bitmap using the Boolean OR operator.
NOTSRCCOPY	Copies the inverted source bitmap to the destination.
NOTSRCERASE	Inverts the result of combining the destination and source bitmaps using the Boolean OR operator.
PATCOPY	Copies the pattern to the destination bitmap.
PATINVERT	Combines the destination bitmap with the pattern using the Boolean XOR operator.
PATPAINT	Combines the inverted source bitmap with the pattern using the Boolean OR operator. Combines the result of this operation with the destination bitmap using the Boolean OR operator.
SRCAND	Combines pixels of the destination and source bitmaps using the Boolean AND operator.
SRCCOPY	Copies the source bitmap to the destination bitmap.
SRCERASE	Inverts the destination bitmap and combines the result with the source bitmap using the Boolean AND operator.
SRCINVERT	Combines pixels of the destination and source bitmaps using the Boolean XOR operator.
SRCPAINT	Combines pixels of the destination and source bitmaps using the Boolean OR operator.
WHITENESS	Turns all output white.

For a complete list of the raster-operation codes, see Chapter 11, "Binary and Ternary Raster-Operation Codes," in *Reference, Volume 2.*

BringWindowToTop

Syntax void **BringWindowToTop**(*hWnd*)

This function brings a pop-up or child window to the top of a stack of overlapping windows. In addition, it activates pop-up and top-level windows. The **BringWindowTo-Top** function should be used to uncover any window that is partially or completely obscured by any overlapping windows.

Parameter	Type/Description
hWnd	**HWND** Identifies the pop-up or child window that is to be brought to the top.

Return Value None.

BuildCommDCB

Syntax int **BuildCommDCB**(*lpDef, lpDCB*)

This function translates the definition string specified by the *lpDef* parameter into appropriate device-control block codes and places these codes into the block pointed to by the *lpDCB* parameter.

Parameter	Type/Description
lpDef	**LPSTR** Points to a null-terminated character string that specifies the device-control information for a device. The string must have the same form as the DOS **MODE** command-line parameter.
lpDCB	**DCB FAR *** Points to the **DCB** data structure that is to receive the translated string. The structure defines the control setting for the serial-communication device.

Return Value The return value specifies the result of the function. It is zero if the string is translated. It is negative if an error occurs.

Comments The **BuildCommDCB** function only fills the buffer. An application should call **SetComm-State** to apply these settings to the port. Also, by default, **BuildCommDCB** specifies Xon/Xoff and hardware flow control as disabled. An application should set the appropriate fields in the **DCB** data structure to enable flow control.

CallMsgFilter

Syntax

BOOL CallMsgFilter(*lpMsg*, *nCode*)

This function passes the given message and code to the current message filter function. The message filter function is an application-specified function that examines and modifies all messages. An application specifies the function by using the **SetWindowsHook** function.

Parameter	Type/Description
lpMsg	**LPMSG** Points to an **MSG** data structure that contains the message to be filtered.
nCode	**int** Specifies a code used by the filter function to determine how to process the message.

Return Value

The return value specifies the state of message processing. It is FALSE if the message should be processed. It is TRUE if the message should not be processed further.

Comments

The **CallMsgFilter** function is usually called by Windows to let applications examine and control the flow of messages during internal processing in menus and scroll bars or when moving or sizing a window.

Values given for the *nCode* parameter must not conflict with any of the MSGF_ and HC_ values passed by Windows to the message filter function.

CallWindowProc

Syntax

LONG CallWindowProc(*lpPrevWndFunc*, *hWnd*, *wMsg*, *wParam*, *lParam*)

This function passes message information to the function specified by the *lpPrevWndFunc* parameter. The **CallWindowProc** function is used for window subclassing. Normally, all windows with the same class share the same window function. A subclass is a window or set of windows belonging to the same window class whose messages are intercepted and processed by another function (or functions) before being passed to the window function of that class.

The **SetWindowLong** function creates the subclass by changing the window function associated with a particular window, causing Windows to call the new window function instead of the previous one. Any messages not processed by the new window function must be passed to the previous window function by calling **CallWindowProc**. This allows a chain of window functions to be created.

Parameter	Type/Description
lpPrevWndFunc	**FARPROC** Is the procedure-instance address of the previous window function.
hWnd	**HWND** Identifies the window that receives the message.
wMsg	**WORD** Specifies the message number.
wParam	**WORD** Specifies additional message-dependent information.
lParam	**DWORD** Specifies additional message-dependent information.

Return Value The return value specifies the result of the message processing. The possible return values depend on the message sent.

Catch

Syntax **int Catch**(*lpCatchBuf*)

This function catches the current execution environment and copies it to the buffer pointed to by the *lpCatchBuf* parameter. The execution environment is the state of all system registers and the instruction counter.

Parameter	Type/Description
lpCatchBuf	**LPCATCHBUF** Points to the **CATCHBUF** structure that will receive the execution environment.

Return Value The return value specifies whether the execution environment is copied to the buffer. It is zero if the environment is copied to the buffer.

Comments The **Throw** function uses the buffer to restore the execution environment to its previous values.

The **Catch** function is similar to the C run-time **setjmp** function (which is incompatible with the Windows environment).

ChangeClipboardChain

Syntax **BOOL ChangeClipboardChain**(*hWnd*, *hWndNext*)

This function removes the window specified by the *hWnd* parameter from the chain of clipboard viewers and makes the window specified by the *hWndNext* parameter the descendant of the *hWnd* parameter's ancestor in the chain.

Parameter	Type/Description
hWnd	**HWND** Identifies the window that is to be removed from the chain. The handle must previously have been passed to the **SetClipboardViewer** function.
hWndNext	**HWND** Identifies the window that follows *hWnd* in the clipboard-viewer chain (this is the handle returned by the **SetClipboardViewer** function, unless the sequence was changed in response to a WM_CHANGECBCHAIN message).

Return Value The return value specifies the status of the *hWnd* window. It is nonzero if the window is found and removed. Otherwise, it is zero.

ChangeMenu

The Microsoft Windows version 3.0 SDK has replaced this function with five specialized functions. These new functions are:

Function	Description
AppendMenu	Appends a menu item to the end of a menu.
DeleteMenu	Deletes a menu item from a menu, destroying the menu item.
InsertMenu	Inserts a menu item into a menu.
ModifyMenu	Modifies a menu item in a menu.
RemoveMenu	Removes a menu item from a menu but does not destroy the menu item.

Applications written for SDK versions 2.1 and earlier may continue to call **ChangeMenu** as previously documented. New applications should call the new functions listed above.

ChangeSelector 3.0

Syntax WORD **ChangeSelector**(*wDestSelector*, *wSourceSelector*)

This function generates a code selector that corresponds to a given data selector, or a data selector that corresponds to a given code selector.

The *wSourceSelector* parameter specifies the selector to be copied and converted; the *wDestSelector* parameter is a selector previously allocated by a call to the **AllocSelector** function. **ChangeSelector** modifies the destination selector to have the same properties as the source selector, but with the opposite code or data attribute. This function changes only the attributes of the selector, not the value of the selector.

Parameter	Type/Description
wDestSelector	**WORD** Specifies a selector previously allocated by **AllocSelector** that receives the converted selector.
wSourceSelector	**WORD** Specifies the selector to be converted.

Return Value The return value is the copied and converted selector. It is zero if the function failed.

Comments Windows does not attempt to track changes to the source selector. Consequently, the application should use the converted destination selector immediately after it is returned by this function before any movement of memory can occur.

An application should not use this function unless it is absolutely necessary. Use of this function violates preferred Windows programming practices.

CheckDlgButton

Syntax void **CheckDlgButton**(*hDlg*, *nIDButton*, *wCheck*)

This function places a checkmark next to or removes a checkmark from a button control, or changes the state of a three-state button. The **CheckDlgButton** function sends a BM_SETCHECK message to the button control that has the specified ID in the given dialog box.

Parameter	Type/Description
hDlg	**HWND** Identifies the dialog box that contains the button.
nIDButton	**int** Specifies the button control to be modified.

Parameter	Type/Description
wCheck	**WORD** Specifies the action to take. If the *wCheck* parameter is nonzero, the **CheckDlgButton** function places a checkmark next to the button; if zero, the checkmark is removed. For three-state buttons, if *wCheck* is 2, the button is grayed; if *wCheck* is 1, it is checked; if *wCheck* is 0, the checkmark is removed.

Return Value None.

CheckMenuItem

Syntax **BOOL CheckMenuItem**(*hMenu*, *wIDCheckItem*, *wCheck*)

This function places checkmarks next to or removes checkmarks from menu items in the pop-up menu specified by the *hMenu* parameter. The *wIDCheckItem* parameter specifies the item to be modified.

Parameter	Type/Description
hMenu	**HMENU** Identifies the menu.
wIDCheck-Item	**WORD** Specifies the menu item to be checked.
wCheck	**WORD** Specifies how to check the menu item and how to determine the item's position in the menu. The *wCheck* parameter can be a combination of the MF_CHECKED or MF_UNCHECKED with MF_BYPOSITION or MF_BYCOMMAND flags. These flags can be combined by using the bitwise OR operator. They have the following meanings:

Value	Meaning
MF_BYCOMMAND	Specifies that the *wIDCheckItem* parameter gives the menu-item ID (MF_BYCOMMAND is the default).
MF_BYPOSITION	Specifies that the *wIDCheckItem* parameter gives the position of the menu item (the first item is at position zero).
MF_CHECKED	Adds checkmark.
MF_UNCHECKED	Removes checkmark.

Return Value The return value specifies the previous state of the item. It is either MF_CHECKED or MF_UNCHECKED. The return value is –1 if the menu item does not exist.

Comments The *wIDCheckItem* parameter may identify a pop-up menu item as well as a menu item. No special steps are required to check a pop-up menu item.

Top-level menu items cannot be checked.

A pop-up menu item should be checked by position since it does not have a menu-item identifier associated with it.

CheckRadioButton

Syntax **void CheckRadioButton**(*hDlg*, *nIDFirstButton*, *nIDLastButton*, *nIDCheckButton*)

This function checks the radio button specified by the *nIDCheckButton* parameter and removes the checkmark from all other radio buttons in the group of buttons specified by the *nIDFirstButton* and *nIDLastButton* parameters. The **CheckRadioButton** function sends a BM_SETCHECK message to the radio-button control that has the specified ID in the given dialog box.

Parameter	Type/Description
hDlg	**HWND** Identifies the dialog box.
nIDFirstButton	**int** Specifies the integer identifier of the first radio button in the group.
nIDLastButton	**int** Specifies the integer identifier of the last radio button in the group.
nIDCheckButton	**int** Specifies the integer identifier of the radio button to be checked.

Return Value None.

ChildWindowFromPoint

Syntax **HWND ChildWindowFromPoint**(*hWndParent*, *Point*)

This function determines which, if any, of the child windows belonging to the given parent window contains the specified point.

Parameter	Type/Description
hWndParent	**HWND** Identifies the parent window.
Point	**POINT** Specifies the client coordinates of the point to be tested.

Return Value The return value identifies the child window that contains the point. It is NULL if the given point lies outside the parent window. If the point is within the parent window but is not contained within any child window, the handle of the parent window is returned.

Chord

Syntax **BOOL Chord**(*hDC*, *X1*, *Y1*, *X2*, *Y2*, *X3*, *Y3*, *X4*, *Y4*)

This function draws a chord (a region bounded by the intersection of an ellipse and a line segment). The (*X1*, *Y1*) and (*X2*, *Y2*) parameters specify the upper-left and lower-right corners, respectively, of a rectangle bounding the ellipse that is part of the chord. The (*X3*, *Y3*) and (*X4*, *Y4*) parameters specify the endpoints of a line that intersects the ellipse. The chord is drawn by using the selected pen and filled by using the selected brush.

Parameter	Type/Description
hDC	**HDC** Identifies the device context in which the chord will appear.
X1	**int** Specifies the *x*-coordinate of the bounding rectangle's upper-left corner.
Y1	**int** Specifies the *y*-coordinate of the bounding rectangle's upper-left corner.
X2	**int** Specifies the *x*-coordinate of the bounding rectangle's lower-right corner.
Y2	**int** Specifies the *y*-coordinate of the bounding rectangle's lower-right corner.
X3	**int** Specifies the *x*-coordinate of one end of the line segment.
Y3	**int** Specifies the *y*-coordinate of one end of the line segment.
X4	**int** Specifies the *x*-coordinate of one end of the line segment.
Y4	**int** Specifies the *y*-coordinate of one end of the line segment.

Return Value The return value specifies whether or not the arc is drawn. It is nonzero if the arc is drawn. Otherwise, it is zero.

ClearCommBreak

Syntax **int ClearCommBreak**(*nCid*)

This function restores character transmission and places the transmission line in a non-break state.

Parameter	Type/Description
nCid	**int** Specifies the communication device to be restored. The **OpenComm** function returns this value.

Return Value The return value specifies the result of the function. It is zero if the function is successful. It is negative if the *nCid* parameter is not a valid device.

ClientToScreen

Syntax **void ClientToScreen**(*hWnd*, *lpPoint*)

This function converts the client coordinates of a given point on the display to screen coordinates. The **ClientToScreen** function uses the client coordinates in the **POINT** data structure, pointed to by the *lpPoint* parameter, to compute new screen coordinates; it then replaces the coordinates in the structure with the new coordinates. The new screen coordinates are relative to the upper-left corner of the system display.

Parameter	Type/Description
hWnd	**HWND** Identifies the window whose client area will be used for the conversion.
lpPoint	**LPPOINT** Points to a **POINT** data structure that contains the client coordinates to be converted.

Return Value None.

Comments The **ClientToScreen** function assumes that the given point is in client coordinates and is relative to the given window.

ClipCursor

Syntax **void** **ClipCursor**(*lpRect*)

This function confines the cursor to the rectangle on the display screen given by the *lpRect* parameter. If a subsequent cursor position, given with the **SetCursorPos** function or the mouse, lies outside the rectangle, Windows automatically adjusts the position to keep the cursor inside. If *lpRect* is NULL, the cursor is free to move anywhere on the display screen.

Parameter	Type/Description
lpRect	**LPRECT** Points to a **RECT** data structure that contains the screen coordinates of the upper-left and lower-right corners of the confining rectangle.

Return Value None.

Comments The cursor is a shared resource. An application that has confined the cursor to a given rectangle must free it before relinquishing control to another application.

CloseClipboard

Syntax **BOOL** **CloseClipboard**( )

This function closes the clipboard. The **CloseClipboard** function should be called when a window has finished examining or changing the clipboard. It lets other applications access the clipboard.

This function has no parameters.

Return Value The return value specifies whether the clipboard is closed. It is nonzero if the clipboard is closed. Otherwise, it is zero.

CloseComm

Syntax **int** **CloseComm**(*nCid*)

This function closes the communication device specified by the *nCid* parameter and frees any memory allocated for the device's transmit and receive queues. All characters in the output queue are sent before the communication device is closed.

Parameter	Type/Description
nCid	**int** Specifies the device to be closed. The **OpenComm** function returns this value.

Return Value The return value specifies the result of the function. It is zero if the device is closed. It is negative if an error occurred.

CloseMetaFile

Syntax **HANDLE CloseMetaFile**(*hDC*)

This function closes the metafile device context and creates a metafile handle that can be used to play the metafile by using the **PlayMetaFile** function.

Parameter	Type/Description
hDC	**HANDLE** Identifies the metafile device context to be closed.

Return Value The return value identifies the metafile if the function is successful. Otherwise, it is NULL.

CloseSound

Syntax **void CloseSound( )**

This function closes access to the play device and frees the device for opening by other applications. The **CloseSound** function flushes all voice queues and frees any buffers allocated for these queues.

This function has no parameters.

Return Value None.

CloseWindow

Syntax **void CloseWindow**(*hWnd*)

This function minimizes the specified window. If the window is an overlapped window, it is minimized by removing the client area and caption of the open window from the display screen and moving the window's icon into the icon area of the screen.

Parameter	Type/Description
hWnd	**HWND** Identifies the window to be minimized.

Return Value None.

Comments This function has no effect if the *hWnd* parameter is a handle to a pop-up or child window.

CombineRgn

Syntax **int CombineRgn**(*hDestRgn*, *hSrcRgn1*, *hSrcRgn2*, *nCombineMode*)

This function creates a new region by combining two existing regions. The method used to combine the regions is specified by the *nCombineMode* parameter.

Parameter	Type/Description
hDestRgn	**HRGN** Identifies an existing region that will be replaced by the new region.
hSrcRgn1	**HRGN** Identifies an existing region.
hSrcRgn2	**HRGN** Identifies an existing region.
nCombineMode	**int** Specifies the operation to be performed on the two existing regions. It can be any one of the following values:

Value	Meaning
RGN_AND	Uses overlapping areas of both regions (intersection).
RGN_COPY	Creates a copy of region 1 (identified by *hSrcRgn1*).
RGN_DIFF	Saves the areas of region 1 (identified by the *hSrcRgn1* parameter) that are not part of region 2 (identified by the *hSrcRgn2* parameter).
RGN_OR	Combines all of both regions (union).
RGN_XOR	Combines both regions but removes overlapping areas.

Return Value

The return value specifies the type of the resulting region. It can be any one of the following values:

Value	Meaning
COMPLEXREGION	New region has overlapping borders.
ERROR	No new region created.
NULLREGION	New region is empty.
SIMPLEREGION	New region has no overlapping borders.

Comments

If the *hDestRgn* parameter does not identify an existing region, the application must pass a far pointer to a previously allocated **HRGN** as the *hDestRgn* parameter.

CopyMetaFile

Syntax

HANDLE CopyMetaFile(*hSrcMetaFile*, *lpFilename*)

This function copies the source metafile to the file pointed to by the *lpFilename* parameter and returns a handle to the new metafile. If *lpFilename* is NULL, the source is copied to a memory metafile.

Parameter	Type/Description
hSrcMetaFile	**HANDLE** Identifies the source metafile.
lpFilename	**LPSTR** Points to a null-terminated character string that specifies the file that is to receive the metafile.

Return Value

The return value identifies the new metafile.

CopyRect

Syntax

int CopyRect(*lpDestRect*, *lpSourceRect*)

This function copies the rectangle pointed to by the *lpSourceRect* parameter to the **RECT** data structure pointed to by the *lpDestRect* parameter.

Parameter	Type/Description	
lpDestRect	**LPRECT**	Points to a **RECT** data structure.
lpSourceRect	**LPRECT**	Points to a **RECT** data structure.

Return Value Although the **CopyRect** function return type is an integer, the return value is not used and has no meaning.

CountClipboardFormats

Syntax **int CountClipboardFormats( )**

This function retrieves a count of the number of formats the clipboard can render.

This function has no parameters.

Return Value The return value specifies the number of data formats in the clipboard.

CountVoiceNotes

Syntax **int CountVoiceNotes(***nVoice***)**

This function retrieves a count of the number of notes in the specified queue. Only those queue entries that result from calls to the **SetVoiceNote** function are counted.

Parameter	Type/Description	
nVoice	**int**	Specifies the voice queue to be counted. The first voice queue is numbered 1.

Return Value The return value specifies the number of notes in the given queue.

CreateBitmap

Syntax **HBITMAP CreateBitmap(***nWidth, nHeight, nPlanes, nBitCount, lpBits***)**

This function creates a device-dependent memory bitmap that has the specified width, height, and bit pattern. The bitmap can subsequently be selected as the current bitmap for a memory display by using the **SelectObject** function.

Although a bitmap cannot be copied directly to a display device, the **BitBlt** function can copy it from a memory display context (in which it is the current bitmap) to any compatible device.

Parameter	Type/Description
nWidth	**int** Specifies the width (in pixels) of the bitmap.
nHeight	**int** Specifies the height (in pixels) of the bitmap.
nPlanes	**BYTE** Specifies the number of color planes in the bitmap. Each plane has *nWidth* × *nHeight* × *nBitCount* bits.
nBitCount	**BYTE** Specifies the number of color bits per display pixel.
lpBits	**LPSTR** Points to a short-integer array that contains the initial bitmap bit values. If it is NULL, the new bitmap is left uninitialized. For more information, see the description of the **bmBits** field in the **BITMAP** data structure in Chapter 7, "Data Types and Structures," in *Reference, Volume 2*.

Return Value The return value identifies a bitmap if the function is successful. Otherwise, it is NULL.

CreateBitmapIndirect

Syntax **HBITMAP CreateBitmapIndirect**(*lpBitmap*)

This function creates a bitmap that has the width, height, and bit pattern given in the data structure pointed to by the *lpBitmap* parameter. Although a bitmap cannot be directly selected for a display device, it can be selected as the current bitmap for a memory display and copied to any compatible display device by using the **BitBlt** function.

Parameter	Type/Description
lpBitmap	**BITMAP FAR *** Points to a **BITMAP** data structure that contains information about the bitmap.

Return Value The return value identifies a bitmap if the function is successful. Otherwise, it is NULL.

CreateBrushIndirect

Syntax **HBRUSH** **CreateBrushIndirect**(*lpLogBrush*)

This function creates a logical brush that has the style, color, and pattern given in the data structure pointed to by the *lpLogBrush* parameter. The brush can subsequently be selected as the current brush for any device.

Parameter	Type/Description
lpLogBrush	**LOGBRUSH FAR** * Points to a **LOGBRUSH** data structure that contains information about the brush.

Return Value The return value identifies a logical brush if the function is successful. Otherwise, it is NULL.

Comments A brush created using a monochrome (one plane, one bit per pixel) bitmap is drawn using the current text and background colors. Pixels represented by a bit set to 0 will be drawn with the current text color, and pixels represented by a bit set to 1 will be drawn with the current background color.

CreateCaret

Syntax **void** **CreateCaret**(*hWnd*, *hBitmap*, *nWidth*, *nHeight*)

This function creates a new shape for the system caret and assigns ownership of the caret to the given window. The caret shape can be a line, block, or bitmap as defined by the *hBitmap* parameter. If *hBitmap* is a bitmap handle, the *nWidth* and *nHeight* parameters are ignored; the bitmap defines its own width and height. (The bitmap handle must have been previously created by using the **CreateBitmap**, **CreateDIBitmap**, or **LoadBitmap** function.) If *hBitmap* is NULL or 1, *nWidth* and *nHeight* give the caret's width and height (in logical units); the exact width and height (in pixels) depend on the window's mapping mode.

If *nWidth* or *nHeight* is zero, the caret width or height is set to the system's window-border width or height. Using the window-border width or height guarantees that the caret will be visible on a high-resolution display.

The **CreateCaret** function automatically destroys the previous caret shape, if any, regardless of which window owns the caret. Once created, the caret is initially hidden. To show the caret, the **ShowCaret** function must be called.

Parameter	Type/Description
hWnd	**HWND** Identifies the window that owns the new caret.
hBitmap	**HBITMAP** Identifies the bitmap that defines the caret shape. If *hBitmap* is NULL, the caret is solid; if *hBitmap* is 1, the caret is gray.
nWidth	**int** Specifies the width of the caret (in logical units).
nHeight	**int** Specifies the height of the caret (in logical units).

Return Value None.

Comments The system caret is a shared resource. A window should create a caret only when it has the input focus or is active. It should destroy the caret before losing the input focus or becoming inactive.

The system's window-border width or height can be retrieved by using the **GetSystemMetrics** function with the SM_CXBORDER and SM_CYBORDER indexes.

CreateCompatibleBitmap

Syntax **HBITMAP CreateCompatibleBitmap**(*hDC, nWidth, nHeight*)

This function creates a bitmap that is compatible with the device specified by the *hDC* parameter. The bitmap has the same number of color planes or the same bits-per-pixel format as the specified device. It can be selected as the current bitmap for any memory device that is compatible with the one specified by *hDC*.

If *hDC* is a memory device context, the bitmap returned has the same format as the currently selected bitmap in that device context. A memory device context is a block of memory that represents a display surface. It can be used to prepare images in memory before copying them to the actual display surface of the compatible device.

When a memory device context is created, GDI automatically selects a monochrome stock bitmap for it.

Since a color memory device context can have either color or monochrome bitmaps selected, the format of the bitmap returned by the **CreateCompatibleBitmap** function is not always the same; however, the format of a compatible bitmap for a nonmemory device context is always in the format of the device.

Parameter	Type/Description
hDC	**HDC** Identifies the device context.
nWidth	**int** Specifies the width (in bits) of the bitmap.
nHeight	**int** Specifies the height (in bits) of the bitmap.

Return Value The return value identifies a bitmap if the function is successful. Otherwise, it is NULL.

CreateCompatibleDC

Syntax **HDC CreateCompatibleDC**(*hDC*)

This function creates a memory device context that is compatible with the device specified by the *hDC* parameter. A memory device context is a block of memory that represents a display surface. It can be used to prepare images in memory before copying them to the actual device surface of the compatible device.

When a memory device context is created, GDI automatically selects a 1-by-1 monochrome stock bitmap for it.

Parameter	Type/Description
hDC	**HDC** Identifies the device context. If *hDC* is NULL, the function creates a memory device context that is compatible with the system display.

Return Value The return value identifies the new memory device context if the function is successful. Otherwise, it is NULL.

Comments This function can only be used to create compatible device contexts for devices that support raster operations. For more information, see the RC_BITBLT raster capability in the **GetDeviceCaps** function, later in this chapter.

GDI output functions can be used with a memory device context only if a bitmap has been created and selected into that context.

When the application no longer requires the device context, it should free it by calling the **DeleteDC** function.

CreateCursor 3.0

Syntax **HCURSOR** **CreateCursor**(*hInstance*, *nXhotspot*, *nYhotspot*, *nWidth*, *nHeight*, *lpANDbitPlane*, *lpXORbitPlane*)

This function creates a cursor that has specified width, height, and bit patterns.

Parameter	Type/Description
hInstance	**HANDLE** Identifies an instance of the module creating the cursor.
nXhotspot	**int** Specifies the horizontal position of the cursor hotspot.
nYhotspot	**int** Specifies the vertical position of the cursor hotspot.
nWidth	**int** Specifies the width in pixels of the cursor.
nHeight	**int** Specifies the height in pixels of the cursor.
lpANDbitPlane	**LPSTR** Points to an array of bytes containing the bit values for the AND mask of the cursor. This can be the bits of a device-dependent monochrome bitmap.
lpXORbitPlane	**LPSTR** Points to an array of bytes containing the bit values for the XOR mask of the cursor. This can be the bits of a device-dependent monochrome bitmap.

Return Value The return value identifies the cursor if the function was successful. Otherwise, it is NULL.

CreateDC

Syntax **HDC** **CreateDC**(*lpDriverName*, *lpDeviceName*, *lpOutput*, *lpInitData*)

This function creates a device context for the specified device. The *lpDriverName*, *lpDeviceName*, and *lpOutput* parameters specify the device driver, device name, and physical output medium (file or port), respectively.

Parameter	Type/Description
lpDriverName	**LPSTR** Points to a null-terminated character string that specifies the DOS filename (without extension) of the device driver (for example, Epson ®).

Parameter	Type/Description
lpDeviceName	**LPSTR** Points to a null-terminated character string that specifies the name of the specific device to be supported (for example, Epson FX-80). The *lpDeviceName* parameter is used if the module supports more than one device.
lpOutput	**LPSTR** Points to a null-terminated character string that specifies the DOS file or device name for the physical output medium (file or output port).
lpInitData	**LPDEVMODE** Points to a **DEVMODE** data structure containing device-specific initialization data for the device driver. The **ExtDeviceMode** retrieves this structure filled in for a given device. The *lpInitData* parameter must be NULL if the device driver is to use the default initialization (if any) specified by the user through the Control Panel.

Return Value

The return value identifies a device context for the specified device if the function is successful. Otherwise, it is NULL.

Comments

DOS device names follow DOS conventions; an ending colon (:) is recommended, but optional. Windows strips the terminating colon so that a device name ending with a colon is mapped to the same port as the same name without a colon. The driver and port names must not contain leading or trailing spaces.

CreateDialog

Syntax

HWND CreateDialog(*hInstance*, *lpTemplateName*, *hWndParent*, *lpDialogFunc*)

This function creates a modeless dialog box that has the size, style, and controls defined by the dialog-box template given by the *lpTemplateName* parameter. The *hWndParent* parameter identifies the application window that owns the dialog box. The dialog function pointed to by the *lpDialogFunc* parameter processes any messages received by the dialog box.

The **CreateDialog** function sends a WM_INITDIALOG message to the dialog function before displaying the dialog box. This message allows the dialog function to initialize the dialog-box controls.

CreateDialog returns immediately after creating the dialog box. It does not wait for the dialog box to begin processing input.

Parameter	Type/Description
hInstance	**HANDLE** Identifies an instance of the module whose executable file contains the dialog-box template.
lpTemplateName	**LPSTR** Points to a character string that names the dialog-box template. The string must be a null-terminated character string.
hWndParent	**HWND** Identifies the window that owns the dialog box.
lpDialogFunc	**FARPROC** Is the procedure-instance address for the dialog function. See the following "Comments" section for details.

Return Value

The return value is the window handle of the dialog box. It is NULL if the function cannot create the dialog box.

Comments

Use the WS_VISIBLE style for the dialog-box template if the dialog box should appear in the parent window upon creation.

Use the **DestroyWindow** function to destroy a dialog box created by the **CreateDialog** function.

A dialog box can contain up to 255 controls.

The callback function must use the Pascal calling convention and must be declared **FAR**.

Callback Function

BOOL FAR PASCAL *DialogFunc*(*hDlg*, *wMsg*, *wParam*, *lParam*)
HWND *hDlg*;
WORD *wMsg*;
WORD *wParam*;
DWORD *lParam*;

DialogFunc is a placeholder for the application-supplied function name. The actual name must be exported by including it in an **EXPORTS** statement in the application's module-definition file.

Parameter	Definition
hDlg	Identifies the dialog box that receives the message.
wMsg	Specifies the message number.
wParam	Specifies 16 bits of additional message-dependent information.
lParam	Specifies 32 bits of additional message-dependent information.

Return Value

Except in response to the WM_INITDIALOG message, the dialog function should return nonzero if the function processes the message, and zero if it does not. In response to a WM_INITDIALOG message, the dialog function should return zero if it calls the **SetFocus** function to set the focus to one of the controls in the dialog box. Otherwise, it should return nonzero, in which case Windows will set the focus to the first control in the dialog box that can be given the focus.

Comments

The dialog function is used only if the dialog class is used for the dialog box. This is the default class and is used if no explicit class is given in the dialog-box template. Although the dialog function is similar to a window function, it must not call the **DefWindowProc** function to process unwanted messages. Unwanted messages are processed internally by the dialog-class window function.

The dialog-function address, passed as the *lpDialogFunc* parameter, must be created by using the **MakeProcInstance** function.

CreateDialogIndirect

Syntax

HWND CreateDialogIndirect(*hInstance*, *lpDialogTemplate*, *hWndParent*, *lpDialogFunc*)

This function creates a modeless dialog box that has the size, style, and controls defined by the dialog-box template given by the *lpDialogTemplate* parameter. The *hWndParent* parameter identifies the application window that owns the dialog box. The dialog function pointed to by the *lpDialogFunc* parameter processes any messages received by the dialog box.

The **CreateDialogIndirect** function sends a WM_INITDIALOG message to the dialog function before displaying the dialog box. This message allows the dialog function to initialize the dialog-box controls.

CreateDialogIndirect returns immediately after creating the dialog box. It does not wait for the dialog box to begin processing input.

Parameter	Type/Description
hInstance	**HANDLE** Identifies an instance of the module whose executable file contains the dialog-box template.
lpDialogTemplate	**LPSTR** Points to a block of memory that contains a **DLGTEMPLATE** data structure.
hWndParent	**HWND** Identifies the window that owns the dialog box.

Parameter	Type/Description
lpDialogFunc	**FARPROC** Is the procedure-instance address of the dialog function. See the following "Comments" section for details.

Return Value

The return value is the window handle of the dialog box. It is NULL if the function cannot create either the dialog box or any controls in the dialog box.

Comments

Use the WS_VISIBLE style in the dialog-box template if the dialog box should appear in the parent window upon creation.

A dialog box can contain up to 255 controls.

The callback function must use the Pascal calling convention and must be declared **FAR**.

Callback Function

BOOL FAR PASCAL *DialogFunc*(*hDlg*, *wMsg*, *wParam*, *lParam*)
HWND *hDlg*;
WORD *wMsg*;
WORD *wParam*;
DWORD *lParam*;

DialogFunc is a placeholder for the application-supplied function name. The actual name must be exported by including it in an **EXPORTS** statement in the application's module-definition file.

Parameter	Definition
hDlg	Identifies the dialog box that receives the message.
wMsg	Specifies the message number.
wParam	Specifies 16 bits of additional message-dependent information.
lParam	Specifies 32 bits of additional message-dependent information.

Return Value

Except in response to the WM_INITDIALOG message, the dialog function should return nonzero if the function processes the message, and zero if it does not. In response to a WM_INITDIALOG message, the dialog function should return zero if it calls the **SetFocus** function to set the focus to one of the controls in the dialog box. Otherwise, it should return nonzero, in which case Windows will set the focus to the first control in the dialog box that can be given the focus.

Comments

The dialog function is used only if the dialog class is used for the dialog box. This is the default class and is used if no explicit class is given in the dialog-box template. Although the dialog function is similar to a window function, it must not call the **DefWindowProc** function to process unwanted messages. Unwanted messages are processed internally by the dialog-class window function.

The dialog-function address, passed as the *lpDialogFunc* parameter, must be created by using the **MakeProcInstance** function.

CreateDialogIndirectParam 3.0

Syntax

HWND CreateDialogIndirectParam(*hInstance*, *lpDialogTemplate*, *hWndParent*, *lpDialogFunc*, *dwInitParam*)

This function creates a modeless dialog box, sends a WM_INITDIALOG message to the dialog function before displaying the dialog box, and passes *dwInitParam* as the message *lParam*. This message allows the dialog function to initialize the dialog-box controls. Otherwise, this function is identical to the **CreateDialogIndirect** function.

For more information on creating a modeless dialog box, see the description of the **CreateDialogIndirect** function.

Parameter	Type/Description
hInstance	**HANDLE** Identifies an instance of the module whose executable file contains the dialog-box template.
lpDialogTemplate	**LPSTR** Points to a block of memory that contains a **DLGTEMPLATE** data structure.
hWndParent	**HWND** Identifies the window that owns the dialog box.
lpDialogFunc	**FARPROC** Is the procedure-instance address of the dialog function. For details, see the "Comments" section in the description of the **CreateDialogIndirect** function.
dwInitParam	**DWORD** Is a 32-bit value which **CreateDialogIndirectParam** passes to the dialog function when it creates the dialog box.

Return Value

The return value is the window handle of the dialog box. It is NULL if the function cannot create either the dialog box or any controls in the dialog box.

CreateDialogParam 3.0

Syntax **HWND CreateDialogParam**(*hInstance*, *lpTemplateName*, *hWndParent*, *lpDialogFunc*, *dwInitParam*)

This function creates a modeless dialog box, sends a WM_INITDIALOG message to the dialog function before displaying the dialog box, and passes *dwInitParam* as the message *lParam*. This message allows the dialog function to initialize the dialog-box controls. Otherwise, this function is identical to the **CreateDialog** function.

For more information on creating a modeless dialog box, see the description of the **CreateDialog** function.

Parameter	Type/Description
hInstance	**HANDLE** Identifies an instance of the module whose executable file contains the dialog-box template.
lpTemplateName	**LPSTR** Points to a character string that names the dialog-box template. The string must be a null-terminated character string.
hWndParent	**HWND** Identifies the window that owns the dialog box.
lpDialogFunc	**FARPROC** Is the procedure-instance address for the dialog function. For details, see the "Comments" section of the **CreateDialog** function.
dwInitParam	**DWORD** Is a 32-bit value which **CreateDialogParam** passes to the dialog function when it creates the dialog box.

Return Value The return value is the window handle of the dialog box. It is –1 if the function cannot create the dialog box.

CreateDIBitmap 3.0

Syntax **HBITMAP CreateDIBitmap**(*hDC*, *lpInfoHeader*, *dwUsage*, *lpInitBits*, *lpInitInfo*, *wUsage*)

This function creates a device-specific memory bitmap from a device-independent bitmap (DIB) specification and optionally sets bits in the bitmap.

Parameter	Type/Description
hDC	**HDC** Identifies the device context.
lpInfoHeader	**LPBITMAPINFOHEADER** Points to a **BITMAPINFO-HEADER** structure that describes the size and format of the device-independent bitmap.
dwUsage	**DWORD** Indicates whether the memory bitmap is to be initialized. If *dwUsage* is set to CBM_INIT, **CreateDIBitmap** will initialize the bitmap with the bits specified by *lpInitBits* and *lpInitInfo*.
lpInitBits	**LPSTR** Points to a byte array that contains the initial bitmap values. The format of the bitmap values depends on the **biBitCount** field of the **BITMAPINFO** structure identified by *lpInitInfo*. See the description of the **BITMAPINFO** data structure in Chapter 7, "Data Types and Structures," in *Reference, Volume 2*, for more information.
lpInitInfo	**LPBITMAPINFO** Points to a **BITMAPINFO** data structure that describes the dimensions and color format of *lpInitBits*.
wUsage	**WORD** Specifies whether the **bmiColors[ ]** fields of the *lpInitInfo* data structure contain explicit RGB values or indexes into the currently realized logical palette. The *wUsage* parameter must be one of the following values:

Value	Meaning
DIB_PAL_COLORS	The color table consists of an array of 16-bit indexes into the currently realized logical palette.
DIB_RGB_COLORS	The color table contains literal RGB values.

Return Value

The return value identifies a bitmap if the function is successful. Otherwise, it is NULL.

Comments

This function also accepts a device-independent bitmap specification formatted for Microsoft OS/2 Presentation Manager versions 1.1 and 1.2 if the *lpInfoHeader* points to a **BITMAPCOREHEADER** data structure and the *lpInitInfo* parameter points to a **BITMAPCOREINFO** data structure.

CreateDIBPatternBrush 3.0

Syntax **HBRUSH CreateDIBPatternBrush**(*hPackedDIB*, *wUsage*)

This function creates a logical brush that has the pattern specified by the device-independent bitmap (DIB) defined by the the *hPackedDIB* parameter. The brush can subsequently be selected for any device that supports raster operations. For more information, see the RC_BITBLT raster capability in the **GetDeviceCaps** function, later in this chapter.

Parameter	Type/Description
hPackedDIB	**GLOBALHANDLE** Identifies a global memory object containing a packed device-independent bitmap. To obtain this handle, an application calls the **GlobalAlloc** function to allocate a block of global memory and then fills the memory with the packed DIB. A packed DIB consists of a **BITMAPINFO** data structure immediately followed by the array of bytes which define the pixels of the bitmap.
wUsage	**WORD** Specifies whether the **bmiColors[ ]** fields of the **BITMAPINFO** data structure contain explicit RGB values or indexes into the currently realized logical palette. The *wUsage* parameter must be one of the following values:

Value	Meaning
DIB_PAL_COLORS	The color table contains literal RGB values. into the currently realized logical palette.
DIB_RGB_COLORS	The color table consists of an array of 16-bit indexes.

Return Value The return value identifies a logical brush if the function is successful. Otherwise, it is NULL.

CreateDiscardableBitmap

Syntax **HBITMAP CreateDiscardableBitmap**(*hDC*, *nWidth*, *nHeight*)

This function creates a discardable bitmap that is compatible with the device identified by the *hDC* parameter. The bitmap has the same number of color planes or the same bits-per-pixel format as the specified device. An application can select this bitmap as the current bitmap for a memory device that is compatible with the one specified by the *hDC* parameter.

Parameter	Type/Description
hDC	**HDC** Identifies a device context.
nWidth	**int** Specifies the width (in bits) of the bitmap.
nHeight	**int** Specifies the height (in bits) of the bitmap.

Return Value The return value identifies a bitmap if the function is successful. Otherwise, it is NULL.

Comments Windows can discard a bitmap created by this function only if an application has not selected it into a display context. If Windows discards the bitmap when it is not selected and the application later attempts to select it, the **SelectObject** function will return zero. When this occurs, the application should remove the handle to the bitmap by using **DeleteObject**.

CreateEllipticRgn

Syntax **HRGN CreateEllipticRgn**(*X1*, *Y1*, *X2*, *Y2*)

This function creates an elliptical region.

Parameter	Type/Description
X1	**int** Specifies the *x*-coordinate of the upper-left corner of the bounding rectangle of the ellipse.
Y1	**int** Specifies the *y*-coordinate of the upper-left corner of the bounding rectangle of the ellipse.
X2	**int** Specifies the *x*-coordinate of the lower-right corner of the bounding rectangle of the ellipse.
Y2	**int** Specifies the *y*-coordinate of the lower-right corner of the bounding rectangle of the ellipse.

Return Value The return value identifies a new region if the function is successful. Otherwise, it is NULL.

Comments The width of the rectangle, specified by the absolute value of $X2 - X1$, must not exceed 32,767 units. This limit also applies to the height of the rectangle.

CreateEllipticRgnIndirect

Syntax **HRGN CreateEllipticRgnIndirect**(*lpRect*)

This function creates an elliptical region.

Parameter	Type/Description
lpRect	**LPRECT** Points to a **RECT** data structure that contains the coordinates of the upper-left and lower-right corners of the bounding rectangle of the ellipse.

Return Value The return value identifies a new region if the function is successful. Otherwise, it is NULL.

Comments The width of the rectangle must not exceed 32,767 units. This limit applies to the height of the rectangle as well.

CreateFont

Syntax **HFONT CreateFont**(*nHeight*, *nWidth*, *nEscapement*, *nOrientation*, *nWeight*, *cItalic*, *cUnderline*, *cStrikeOut*, *cCharSet*, *cOutputPrecision*, *cClipPrecision*, *cQuality*, *cPitchAndFamily*, *lpFacename*)

This function creates a logical font that has the specified characteristics. The logical font can subsequently be selected as the font for any device.

Parameter	Type/Description
nHeight	**int** Specifies the desired height (in logical units) of the font. The font height can be specified in three ways: If *nHeight* is greater than zero, it is transformed into device units and matched against the cell height of the available fonts. If it is zero, a reasonable default size is used. If it is less than zero, it is transformed into device units and the absolute value is matched against the character height of the available fonts. For all height comparisons, the font mapper looks for the largest font that does not exceed the requested size, and, if there is no such font, looks for the smallest font available.

Parameter	Type/Description
nWidth	**int** Specifies the average width (in logical units) of characters in the font. If *nWidth* is zero, the aspect ratio of the device will be matched against the digitization aspect ratio of the available fonts to find the closest match, determined by the absolute value of the difference.
nEscapement	**int** Specifies the angle (in tenths of degrees) of each line of text written in the font (relative to the bottom of the page).
nOrientation	**int** Specifies the angle (in tenths of degrees) of each character's baseline (relative to the bottom of the page).
nWeight	**int** Specifies the desired weight of the font in the range 0 to 1000 (for example, 400 is normal, 700 is bold). If *nWeight* is zero, a default weight is used.
cItalic	**BYTE** Specifies whether the font is italic.
cUnderline	**BYTE** Specifies whether the font is underlined.
cStrikeOut	**BYTE** Specifies whether characters in the font are struck out.
cCharSet	**BYTE** Specifies the desired character set. The following values are predefined: ANSI_CHARSET OEM_CHARSET SYMBOL_CHARSET The OEM character set is system-dependent. Fonts with other character sets may exist in the system. If an application uses a font with an unknown character set, it should not attempt to translate or interpret strings that are to be rendered with that font. Instead, the strings should be passed directly to the output device driver.
cOutputPrecision	**BYTE** Specifies the desired output precision. The output precision defines how closely the output must match the requested font's height, width, character orientation, escapement, and pitch. It can be any one of the following values: OUT_CHARACTER_PRECIS OUT_DEFAULT_PRECIS OUT_STRING_PRECIS OUT_STROKE_PRECIS

Parameter	Type/Description
cClipPrecision	**BYTE** Specifies the desired clipping precision. The clipping precision defines how to clip characters that are partially outside the clipping region. It can be any one of the following values: CLIP_CHARACTER_PRECIS CLIP_DEFAULT_PRECIS CLIP_STROKE_PRECIS
cQuality	**BYTE** Specifies the desired output quality. The output quality defines how carefully GDI must attempt to match the logical-font attributes to those of an actual physical font. It can be any one of the following values: DEFAULT_QUALITY DRAFT_QUALITY PROOF_QUALITY
cPitchAndFamily	**BYTE** Specifies the pitch and family of the font. The two low-order bits specify the pitch of the font and can be any one of the following values: DEFAULT_PITCH FIXED_PITCH VARIABLE_PITCH The four high-order bits of the field specify the font family and can be any one of the following values: FF_DECORATIVE FF_DONTCARE FF_MODERN FF_ROMAN FF_SCRIPT FF_SWISS
lpFacename	**LPSTR** Points to a null-terminated character string that specifies the typeface name of the font. The length of this string must not exceed 30 characters. The **EnumFonts** function can be used to enumerate the typeface names of all currently available fonts.

Return Value The return value identifies a logical font if the function is successful. Otherwise, it is NULL.

Comments The **CreateFont** function does not create a new font. It merely selects the closest match from the fonts available in GDI's pool of physical fonts.

CreateFontIndirect

Syntax **HFONT CreateFontIndirect**(*lpLogFont*)

This function creates a logical font that has the characteristics given in the data structure pointed to by the *lpLogFont* parameter. The font can subsequently be selected as the current font for any device.

Parameter	Type/Description
lpLogFont	**LOGFONT FAR *** Points to a **LOGFONT** data structure that defines the characteristics of the logical font.

Return Value The return value identifies a logical font if the function is successful. Otherwise, it is NULL.

Comments The **CreateFontIndirect** function creates a logical font that has all the specified characteristics. When the font is selected by using the **SelectObject** function, GDI's font mapper attempts to match the logical font with an existing physical font. If it fails to find an exact font, it provides an alternate whose characteristics match as many of the requested characteristics as possible. For a description of the font mapper, see Chapter 2, "Graphics Device Interface Functions."

CreateHatchBrush

Syntax **HBRUSH CreateHatchBrush**(*nIndex*, *crColor*)

This function creates a logical brush that has the specified hatched pattern and color. The brush can subsequently be selected as the current brush for any device.

Parameter	Type/Description
nIndex	**int** Specifies the hatch style of the brush. It can be any one of the following values:

Value	Meaning
HS_BDIAGONAL	45-degree downward hatch (left to right)
HS_CROSS	Horizontal and vertical crosshatch
HS_DIAGCROSS	45-degree crosshatch
HS_FDIAGONAL	45-degree upward hatch (left to right)
HS_HORIZONTAL	Horizontal hatch
HS_VERTICAL	Vertical hatch

Parameter	Type/Description
crColor	**COLORREF** Specifies the foreground color of the brush (the color of the hatches).

Return Value The return value identifies a logical brush if the function is successful. Otherwise, it is NULL.

CreateIC

Syntax **HDC CreateIC**(*lpDriverName*, *lpDeviceName*, *lpOutput*, *lpInitData*)

This function creates an information context for the specified device. The information context provides a fast way to get information about the device without creating a device context.

Parameter	Type/Description
lpDriverName	**LPSTR** Points to a null-terminated character string that specifies the DOS filename (without extension) of the device driver (for example, EPSON).
lpDeviceName	**LPSTR** Points to a null-terminated character string that specifies the name of the specific device to be supported (for example, EPSON FX-80). The *lpDeviceName* parameter is used if the module supports more than one device.

Parameter	Type/Description
lpOutput	**LPSTR** Points to a null-terminated character string that specifies the DOS file or device name for the physical output medium (file or port).
lpInitData	**LPSTR** Points to device-specific initialization data for the device driver. The *lpInitData* parameter must be NULL if the device driver is to use the default initialization (if any) specified by the user through the Control Panel.

Return Value The return value identifies an information context for the specified device if the function is successful. Otherwise, it is NULL.

Comments DOS device names follow DOS conventions; an ending colon (:) is recommended, but optional. Windows strips the terminating colon so that a device name ending with a colon is mapped to the same port as the same name without a colon.

The driver and port names must not contain leading or trailing spaces.

GDI output functions cannot be used with information contexts.

CreateIcon 3.0

Syntax **HICON CreateIcon**(*hInstance*, *nWidth*, *nHeight*, *nPlanes*, *nBitsPixel*, *lpANDbits*, *lpXORbits*)

This function creates an icon that has specified width, height, colors, and bit patterns.

Parameter	Type/Description
hInstance	**HANDLE** Identifies an instance of the module creating the icon.
nWidth	**int** Specifies the width in pixels of the icon.
nHeight	**int** Specifies the height in pixels of the icon.
nPlanes	**BYTE** Specifies the number of planes in the XOR mask of the icon.
nBitsPixel	**BYTE** Specifies the number of bits per pixel in the XOR mask of the icon.

Parameter	Type/Description
lpANDbits	**LPSTR** Points to an array of bytes that contains the bit values for the AND mask of the icon. This array must specify a monochrome mask.
lpXORbits	**LPSTR** Points to an array of bytes that contains the bit values for the XOR mask of the icon. This can be the bits of a monochrome or device-dependent color bitmap.

Return Value The return value identifies an icon if the function is successful. Otherwise, it is NULL.

CreateMenu

Syntax **HMENU CreateMenu( )**

This function creates a menu. The menu is initially empty, but can be filled with menu items by using the **AppendMenu** or **InsertMenu** function.

This function has no parameters.

Return Value The return value identifies the newly created menu. It is NULL if the menu cannot be created.

CreateMetaFile

Syntax **HANDLE CreateMetaFile(***lpFilename***)**

This function creates a metafile device context.

Parameter	Type/Description
lpFilename	**LPSTR** Points to a null-terminated character string that specifies the name of the metafile. If the *lpFilename* parameter is NULL, a device context for a memory metafile is returned.

Return Value The return value identifies a metafile device context if the function is successful. Otherwise, it is NULL.

CreatePalette [3.0]

Syntax **HPALETTE CreatePalette**(*lpLogPalette*)

This function creates a logical color palette.

Parameter	Type/Description
lpLogPalette	**LPLOGPALETTE** Points to a **LOGPALETTE** data structure that contains information about the colors in the logical palette.

Return Value The return value identifies a logical palette if the function was successful. Otherwise, it is NULL.

CreatePatternBrush

Syntax **HBRUSH CreatePatternBrush**(*hBitmap*)

This function creates a logical brush that has the pattern specified by the *hBitmap* parameter. The brush can subsequently be selected for any device that supports raster operations. For more information, see the RC_BITBLT raster capability in the **GetDeviceCaps** function, later in this chapter.

Parameter	Type/Description
hBitmap	**HBITMAP** Identifies the bitmap. It is assumed to have been created by using the **CreateBitmap**, **CreateBitmapIndirect**, **LoadBitmap**, or **CreateCompatibleBitmap** function. The minimum size for a bitmap to be used in a fill pattern is 8-by-8.

Return Value The return value identifies a logical brush if the function is successful. Otherwise, it is NULL.

Comments A pattern brush can be deleted without affecting the associated bitmap by using the **DeleteObject** function. This means the bitmap can be used to create any number of pattern brushes.

A brush created using a monochrome (one plane, one bit per pixel) bitmap is drawn using the current text and background colors. Pixels represented by a bit set to 0 will be drawn with the current text color, and pixels represented by a bit set to 1 will be drawn with the current background color.

CreatePen

Syntax

HPEN CreatePen(*nPenStyle*, *nWidth*, *crColor*)

This function creates a logical pen having the specified style, width, and color. The pen can be subsequently selected as the current pen for any device.

Parameter	Type/Description
nPenStyle	**int** Specifies the pen style. It can be any one of the following values:

Pen Style	Value
PS_SOLID	0
PS_DASH	1
PS_DOT	2
PS_DASHDOT	3
PS_DASHDOTDOT	4
PS_NULL	5
PS_INSIDEFRAME	6

If the width of the pen is greater than 1 and the pen style is PS_INSIDEFRAME, the line is drawn inside the frame of all primitives except polygons and polylines; the pen is drawn with a logical (dithered) color if the pen color does not match an available RGB value. The PS_INSIDEFRAME style is identical to PS_SOLID if the pen width is less than or equal to 1.

Parameter	Type/Description
nWidth	**int** Specifies the width of the pen (in logical units).
crColor	**COLORREF** Specifies the color of the pen.

Return Value

The return value identifies a logical pen if the function is successful. Otherwise, it is NULL.

Comments

Pens with a physical width greater than one pixel will always have either null or solid style or will be dithered if the pen style is PS_INSIDEFRAME.

CreatePenIndirect

Syntax **HPEN CreatePenIndirect**(*lpLogPen*)

This function creates a logical pen that has the style, width, and color given in the data structure pointed to by the *lpLogPen* parameter.

Parameter	Type/Description
lpLogPen	**LOGPEN FAR *** Points to the **LOGPEN** data structure that contains information about the logical pen.

Return Value The return value identifies a logical pen object if the function is successful. Otherwise, it is NULL.

Comments Pens with a physical width greater than one pixel will always have either null or solid style or will be dithered if the pen style is PS_INSIDEFRAME.

CreatePolygonRgn

Syntax **HRGN CreatePolygonRgn**(*lpPoints*, *nCount*, *nPolyFillMode*)

This function creates a polygonal region.

Parameter	Type/Description
lpPoints	**LPPOINT** Points to an array of **POINT** data structures. Each point specifies the *x*- and *y*-coordinates of one vertex of the polygon.
nCount	**int** Specifies the number of points in the array.
nPolyFillMode	**int** Specifies the polygon-filling mode to be used for filling the region. It can be ALTERNATE or WINDING (for an explanation of these modes, see the **SetPolyFillMode** function, later in this chapter).

Return Value The return value identifies a new region if the function is successful. Otherwise, it is NULL.

CreatePolyPolygonRgn [3.0]

Syntax HRGN **CreatePolyPolygonRgn**(*lpPoints*, *lpPolyCounts*, *nCount*, *nPolyFillMode*)

This function creates a region consisting of a series of closed polygons. The region is filled using the mode specified by the *nPolyFillMode* parameter. The polygons may overlap, but they do not have to overlap.

Parameter	Type/Description
lpPoints	**LPPOINT** Points to an array of **POINT** data structures that define the vertices of the polygons. Each polygon must be a closed polygon. The polygons are not automatically closed. The polygons are specified consecutively.
lpPolyCounts	**LPINT** Points to an array of integers, each of which specifies the number of points in one of the polygons in the *lpPoints* array.
nCount	**int** Specifies the total number of integers in the *lpPolyCounts* array.
nPolyFillMode	**int** Specifies the filling mode for the region. The *nPolyFill-Mode* parameter may be either of the following values:

Value	Meaning
ALTERNATE	Selects alternate mode.
WINDING	Selects winding number mode.

Return Value The return value identifies the region if the function was successfull. Otherwise, it is NULL.

Comments In general, the polygon fill modes differ only in cases where a complex, overlapping polygon must be filled (for example, a five-sided polygon that forms a five-pointed star with a pentagon in the center). In such cases, ALTERNATE mode fills every other enclosed region within the polygon (that is, the points of the star), but WINDING mode fills all regions (that is, the points and the pentagon).

When the filling mode is ALTERNATE, GDI fills the area between odd-numbered and even-numbered polygon sides on each scan line. That is, GDI fills the area between the first and second side, between the third and fourth side, and so on.

To fill all parts of the region, WINDING mode causes GDI to compute and draw a border that encloses the region but does not overlap. For example, in WINDING mode, the five-sided polygon that forms the star is computed as a ten-sided polygon with no overlapping sides; the resulting star is filled.

CreatePopupMenu 3.0

Syntax **HMENU CreatePopupMenu( )**

This function creates and returns a handle to an empty pop-up menu.

An application adds items to the pop-up menu by calling **InsertMenu** and **AppendMenu.** The application can add the pop-up menu to an existing menu or pop-up menu, or it may display and track selections on the pop-up menu by calling **TrackPopupMenu.**

This function has no parameters.

Return Value The return value identifies the newly created menu. It is NULL if the menu cannot be created.

CreateRectRgn

Syntax **HRGN CreateRectRgn**(*X1*, *Y1*, *X2*, *Y2*)

This function creates a rectangular region.

Parameter	Type/Description
X1	**int** Specifies the *x*-coordinate of the upper-left corner of the region.
Y1	**int** Specifies the *y*-coordinate of the upper-left corner of the region.
X2	**int** Specifies the *x*-coordinate of the lower-right corner of the region.
Y2	**int** Specifies the *y*-coordinate of the lower-right corner of the region.

Return Value The return value identifies a new region if the function is successful. Otherwise, it is NULL.

Comments The width of the rectangle, specified by the absolute value of $X2 - X1$, must not exceed 32,767 units. This limit applies to the height of the rectangle as well.

CreateRectRgnIndirect

Syntax **HRGN CreateRectRgnIndirect**(*lpRect*)

This function creates a rectangular region.

Parameter	Type/Description
lpRect	**LPRECT** Points to a **RECT** data structure that contains the coordinates of the upper-left and lower-right corners of the region.

Return Value The return value identifies a new region if the function is successful. Otherwise, it is NULL.

Comments The width of the rectangle must not exceed 32,767 units. This limit applies to the height of the rectangle as well.

CreateRoundRectRgn 3.0

Syntax **HRGN CreateRoundRectRgn**(*X1*, *Y1*, *X2*, *Y2*, *X3*, *Y3*)

This function creates a rectangular region with rounded corners.

Parameter	Type/Description
X1	**int** Specifies the *x*-coordinate of the upper-left corner of the region.
Y1	**int** Specifies the *y*-coordinate of the upper-left corner of the region.
X2	**int** Specifies the *x*-coordinate of the lower-right corner of the region.
Y2	**int** Specifies the *y*-coordinate of the lower-right corner of the region.
X3	**int** Specifies the width of the ellipse used to create the rounded corners.
Y3	**int** Specifies the height of the ellipse used to create the rounded corners.

Return Value The return value identifies a new region if the function was successful. Otherwise, it is NULL.

Comments The width of the rectangle, specified by the absolute value of $X2 - X1$, must not exceed 32,767 units. This limit applies to the height of the rectangle as well.

CreateSolidBrush

Syntax **HBRUSH CreateSolidBrush**(*crColor*)

This function creates a logical brush that has the specified solid color. The brush can subsequently be selected as the current brush for any device.

Parameter	Type/Description
crColor	**COLORREF** Specifies the color of the brush.

Return Value The return value identifies a logical brush if the function is successful. Otherwise, it is NULL.

CreateWindow

Syntax **HWND CreateWindow**(*lpClassName, lpWindowName, dwStyle, X, Y, nWidth, nHeight, hWndParent, hMenu, hInstance, lpParam*)

This function creates an overlapped, pop-up, or child window. The **CreateWindow** function specifies the window class, window title, window style, and (optionally) initial position and size of the window. The **CreateWindow** function also specifies the window's parent (if any) and menu.

For overlapped, pop-up, and child windows, the **CreateWindow** function sends WM_CREATE, WM_GETMINMAXINFO, and WM_NCCREATE messages to the window. The *lParam* parameter of the WM_CREATE message contains a pointer to a **CREATESTRUCT** data structure. If WS_VISIBLE style is given, **CreateWindow** sends the window all the messages required to activate and show the window.

If the window style specifies a title bar, the window title pointed to by the *lpWindowName* parameter is displayed in the title bar. When using **CreateWindow** to create controls such as buttons, check boxes, and text controls, the *lpWindowName* parameter specifies the text of the control.

Parameter	Type/Description
lpClassName	**LPSTR** Points to a null-terminated character string that names the window class. The class name can be any name registered with the **RegisterClass** function or any of the predefined control-class names specified in Table 4.2, "Control Classes."
lpWindowName	**LPSTR** Points to a null-terminated character string that represents the window name.
dwStyle	**DWORD** Specifies the style of window being created. It can be any combination of the styles given in Table 4.3, "Window Styles," the control styles given in Table 4.4, "Control Styles," or a combination of styles created by using the bitwise OR operator.
X	**int** Specifies the initial *x*-position of the window. For an overlapped or pop-up window, the *X* parameter is the initial *x*-coordinate of the window's upper-left corner (in screen coordinates). If this value is CW_USEDEFAULT, Windows selects the default position for the window's upper-left corner. For a child window, *X* is the *x*-coordinate of the upper-left corner of the window in the client area of its parent window.
Y	**int** Specifies the initial *y*-position of the window. For an overlapped window, the *Y* parameter is the initial *y*-coordinate of the window's upper-left corner. For a pop-up window, *Y* is the *y*-coordinate (in screen coordinates) of the upper-left corner of the pop-up window. For list-box controls, *Y* is the *y*-coordinate of the upper-left corner of the control's client area. For a child window, *Y* is the *y*-coordinate of the upper-left corner of the child window. All of these coordinates are for the window, not the window's client area.
nWidth	**int** Specifies the width (in device units) of the window. For overlapped windows, the *nWidth* parameter is either the window's width (in screen coordinates) or CW_USEDEFAULT. If *nWidth* is CW_USEDEFAULT, Windows selects a default width and height for the window (the default width extends from the initial *x*-position to the right edge of the screen, and the default height extends from the initial *y*-position to the top of the icon area).

Parameter	Type/Description
nHeight	**int** Specifies the height (in device units) of the window. For overlapped windows, the *nHeight* parameter is the window's height in screen coordinates. If the *nWidth* parameter is CW_USEDEFAULT, Windows ignores *nHeight.*
hWndParent	**HWND** Identifies the parent or owner window of the window being created. A valid window handle must be supplied when creating a child window or an owned window. An owned window is an overlapped window that is destroyed when its owner window is destroyed, hidden when its owner is made iconic, and which is always displayed on top of its owner window. For pop-up windows, a handle can be supplied, but is not required. If the window does not have a parent or is not owned by another window, the **hWndParent** parameter must be set to NULL.
hMenu	**HMENU** Identifies a menu or a child-window identifier. The meaning depends on the window style. For overlapped or pop-up windows, the *hMenu* parameter identifies the menu to be used with the window. It can be NULL, if the class menu is to be used. For child windows, *hMenu* specifies the child-window identifier, an integer value that is used by a dialog-box control to notify its parent of events (such as the EN_HSCROLL message). The child-window identifier is determined by the application and should be unique for all child windows with the same parent window.
hInstance	**HANDLE** Identifies the instance of the module to be associated with the window.
lpParam	**LPSTR** Points to a value that is passed to the window through the **CREATESTRUCT** data structure referenced by the *lParam* parameter of the WM_CREATE message. If an application is calling **CreateWindow** to create a multiple document interface (MDI) client window, *lpParam* must point to a **CLIENTCREATESTRUCT** data structure.

Return Value

The return value identifies the new window. It is NULL if the window is not created.

Comments

For overlapped windows where the *X* parameter is CW_USEDEFAULT, the *Y* parameter can be one of the show-style parameters described with the **ShowWindow** function, or, for the first overlapped window to be created by the application, it can be the *nCmdShow* parameter passed to the WinMain function.

Table 4.2 lists the window control classes; Table 4.3 lists the window styles; Table 4.4 lists the control styles:

Table 4.2 Control Classes

Class	Meaning
BUTTON	Designates a small rectangular child window that represents a button the user can turn on or off by clicking it. Button controls can be used alone or in groups, and can either be labeled or appear without text. Button controls typically change appearance when the user clicks them.
COMBOBOX	Designates a control consisting of a selection field similar to an edit control plus a list box. The list box may be displayed at all times or may be dropped down when the user selects a "pop box" next to the selection field. Depending on the style of the combo box, the user can or cannot edit the contents of the selection field. If the list box is visible, typing characters into the selection box will cause the first list box entry that matches the characters typed to be highlighted. Conversely, selecting an item in the list box displays the selected text in the selection field.
EDIT	Designates a rectangular child window in which the user can enter text from the keyboard. The user selects the control, and gives it the input focus by clicking it or moving to it by using the TAB key. The user can enter text when the control displays a flashing caret. The mouse can be used to move the cursor and select characters to be replaced, or to position the cursor for inserting characters. The BACKSPACE key can be used to delete characters. Edit controls use the variable-pitch system font and display ANSI characters. Applications compiled to run with previous versions of Windows display text with a fixed-pitch system font unless they have been marked by the Windows 3.0 **MARK** utility with the **MEMORY FONT** option. An application can also send the WM_SETFONT message to the edit control to change the default font.

Table 4.2 Control Classes (*continued*)

Class	Meaning
	Edit controls expand tab characters into as many space characters as are required to move the cursor to the next tab stop. Tab stops are assumed to be at every eighth character position.
LISTBOX	Designates a list of character strings. This control is used whenever an application needs to present a list of names, such as filenames, that the user can view and select. The user can select a string by pointing to it and clicking. When a string is selected, it is highlighted and a notification message is passed to the parent window. A vertical or horizontal scroll bar can be used with a list-box control to scroll lists that are too long for the control window. The list box automatically hides or shows the scroll bar as needed.
MDICLIENT	Designates an MDI client window. The MDI client window receives messages which control the MDI application's child windows. The recommended style bits are WS_CLIPCHILDREN and WS_CHILD. To create a scrollable MDI client window which allows the user to scroll MDI child windows into view, an application can also use the WS_HSCROLL and WS_VSCROLL styles.
SCROLLBAR	Designates a rectangle that contains a thumb and has direction arrows at both ends. The scroll bar sends a notification message to its parent window whenever the user clicks the control. The parent window is responsible for updating the thumb position, if necessary. Scroll-bar controls have the same appearance and function as scroll bars used in ordinary windows. Unlike scroll bars, scroll-bar controls can be positioned anywhere in a window and used whenever needed to provide scrolling input for a window. The scroll-bar class also includes size-box controls. A size-box control is a small rectangle that the user can expand to change the size of the window.
STATIC	Designates a simple text field, box, or rectangle that can be used to label, box, or separate other controls. Static controls take no input and provide no output.

Table 4.3 Window Styles

Style	Meaning
DS_LOCALEDIT	Specifies that edit controls in the dialog box will use memory in the application's data segment. By default, all edit controls in dialog boxes use memory outside the application's data segment. This feature may be suppressed by adding the DS_LOCALEDIT flag to the STYLE command for the dialog box. If this flag is not used, EM_GETHANDLE and EM_SETHANDLE messages must not be used since the storage for the control is not in the application's data segment. This feature does not affect edit controls created outside of dialog boxes.
DS_MODALFRAME	Creates a dialog box with a modal dialog-box frame that can be combined with a title bar and System menu by specifying the WS_CAPTION and WS_SYSMENU styles.
DS_NOIDLEMSG	Suppresses WM_ENTERIDLE messages that Windows would otherwise send to the owner of the dialog box while the dialog box is displayed.
DS_SYSMODAL	Creates a system-modal dialog box.
WS_BORDER	Creates a window that has a border.
WS_CAPTION	Creates a window that has a title bar (implies the WS_BORDER style). This style cannot be used with the WS_DLGFRAME style.
WS_CHILD	Creates a child window. Cannot be used with the WS_POPUP style.
WS_CHILDWINDOW	Creates a child window that has the WS_CHILD style.
WS_CLIPCHILDREN	Excludes the area occupied by child windows when drawing within the parent window. Used when creating the parent window.
WS_CLIPSIBLINGS	Clips child windows relative to each other; that is, when a particular child window receives a paint message, the WS_CLIPSIBLINGS style clips all other overlapped child windows out of the region of the child window to be updated. (If WS_CLIPSIBLINGS is not given and child windows overlap, it is possible, when drawing within the client area of a child window, to draw within the client area of a neighboring child window.) For use with the WS_CHILD style only.

Table 4.3 Window Styles (*continued*)

Style	Meaning
WS_DISABLED	Creates a window that is initially disabled.
WS_DLGFRAME	Creates a window with a double border but no title.
WS_GROUP	Specifies the first control of a group of controls in which the user can move from one control to the next by using the DIRECTION keys. All controls defined with the WS_GROUP style after the first control belong to the same group. The next control with the WS_GROUP style ends the style group and starts the next group (that is, one group ends where the next begins). Only dialog boxes use this style.
WS_HSCROLL	Creates a window that has a horizontal scroll bar.
WS_ICONIC	Creates a window that is initially iconic. For use with the WS_OVERLAPPED style only.
WS_MAXIMIZE	Creates a window of maximum size.
WS_MAXIMIZEBOX	Creates a window that has a maximize box.
WS_MINIMIZE	Creates a window of minimum size.
WS_MINIMIZEBOX	Creates a window that has a minimize box.
WS_OVERLAPPED	Creates an overlapped window. An overlapped window has a caption and a border.
WS_OVERLAPPEDWINDOW	Creates an overlapped window having the WS_OVERLAPPED, WS_CAPTION, WS_SYSMENU, WS_THICKFRAME, WS_MINIMIZEBOX, and WS_MAXIMIZEBOX styles.
WS_POPUP	Creates a pop-up window. Cannot be used with the WS_CHILD style.
WS_POPUPWINDOW	Creates a pop-up window that has the WS_BORDER, WS_POPUP, and WS_SYSMENU styles. The WS_CAPTION style must be combined with the WS_POPUPWINDOW style to make the system menu visible.
WS_SYSMENU	Creates a window that has a System-menu box in its title bar. Used only for windows with title bars.
WS_TABSTOP	Specifies one of any number of controls through which the user can move by using the TAB key. The TAB key moves the user to the next control specified by the WS_TABSTOP style. Only dialog boxes use this style.

C

Table 4.3 Window Styles (*continued*)

Style	Meaning
WS_THICKFRAME	Creates a window with a thick frame that can be used to size the window.
WS_VISIBLE	Creates a window that is initially visible. This applies to overlapped and pop-up windows. For overlapped windows, the *Y* parameter is used as a **ShowWindow** function parameter.
WS_VSCROLL	Creates a window that has a vertical scroll bar.

Table 4.4 Control Styles

Style	Meaning
BUTTON Class	
BS_AUTOCHECKBOX	Identical to BS_CHECKBOX, except that the button automatically toggles its state whenever the user clicks it.
BS_AUTORADIOBUTTON	Identical to BS_RADIOBUTTON, except that the button is checked, the application is notified by BN_CLICKED, and the checkmarks are removed from all other radio buttons in the group.
BS_AUTO3STATE	Identical to BS_3STATE, except that the button automatically toggles its state when the user clicks it.
BS_CHECKBOX	Designates a small rectangular button that may be checked; its border is bold when the user clicks the button. Any text appears to the right of the button.
BS_DEFPUSHBUTTON	Designates a button with a bold border. This button represents the default user response. Any text is displayed within the button. Windows sends a message to the parent window when the user clicks the button.
BS_GROUPBOX	Designates a rectangle into which other buttons are grouped. Any text is displayed in the rectangle's upper-left corner.
BS_LEFTTEXT	Causes text to appear on the left side of the radio button or check-box button. Use this style with the BS_CHECKBOX, BS_RADIOBUTTON, or BS_3STATE styles.

Table 4.4 Control Styles (*continued*)

Style	Meaning
BS_OWNERDRAW	Designates an owner-draw button. The parent window is notified when the button is clicked. Notification includes a request to paint, invert, and disable the button.
BS_PUSHBUTTON	Designates a button that contains the given text. The control sends a message to its parent window whenever the user clicks the button.
BS_RADIOBUTTON	Designates a small circular button that can be checked; its border is bold when the user clicks the button. Any text appears to the right of the button. Typically, two or more radio buttons are grouped together to represent mutually exclusive choices, so no more than one button in the group is checked at any time.
BS_3STATE	Identical to BS_CHECKBOX, except that a button can be grayed as well as checked. The grayed state typically is used to show that a check box has been disabled.
COMBOBOX Class	
CBS_AUTOHSCROLL	Automatically scrolls the text in the edit control to the right when the user types a character at the end of the line. If this style is not set, only text which fits within the rectangular boundary is allowed.
CBS_DROPDOWN	Similar to CBS_SIMPLE, except that the list box is not displayed unless the user selects an icon next to the selection field.
CBS_DROPDOWNLIST	Similar to CBS_DROPDOWN, except that the edit control is replaced by a static text item which displays the current selection in the list box.
CBS_HASSTRINGS	An owner-draw combo box contains items consisting of strings. The combo box maintains the memory and pointers for the strings so the application can use the LB_GETTEXT message to retrieve the text for a particular item.

Table 4.4 Control Styles *(continued)*

Style	Meaning
CBS_OEMCONVERT	Text entered in the combo box edit control is converted from the ANSI character set to the OEM character set and then back to ANSI. This ensures proper character conversion when the application calls the **AnsiToOem** function to convert an ANSI string in the combo box to OEM characters. This style is most useful for combo boxes that contain filenames and applies only to combo boxes created with the CBS_SIMPLE or CBS_DROPDOWN styles.
CBS_OWNERDRAWFIXED	The owner of the list box is responsible for drawing its contents; the items in the list box are all the same height.
CBS_OWNERDRAWVARIABLE	The owner of the list box is responsible for drawing its contents; the items in the list box are variable in height.
CBS_SIMPLE	The list box is displayed at all times. The current selection in the list box is displayed in the edit control.
CBS_SORT	Automatically sorts strings entered into the list box.
EDIT Class	
ES_AUTOHSCROLL	Automatically scrolls text to the right by 10 characters when the user types a character at the end of the line. When the user presses the ENTER key, the control scrolls all text back to position zero.
ES_AUTOVSCROLL	Automatically scrolls text up one page when the user presses ENTER on the last line.
ES_CENTER	Centers text in a multiline edit control.
ES_LEFT	Aligns text flush-left.
ES_LOWERCASE	Converts all characters to lowercase as they are typed into the edit control.
ES_MULTILINE	Designates multiple-line edit control. (The default is single-line.) If the ES_AUTOVSCROLL style is specified, the edit control shows as many lines as possible and scrolls vertically when the user presses the ENTER key. If ES_AUTOVSCROLL is not given, the edit control shows as many lines as possible and beeps if ENTER is pressed when no more lines can be displayed.

Table 4.4 Control Styles (*continued*)

Style	Meaning
	If the ES_AUTOHSCROLL style is specified, the multiple-line edit control automatically scrolls horizontally when the caret goes past the right edge of the control. To start a new line, the user must press ENTER. If ES_AUTOHSCROLL is not given, the control automatically wraps words to the beginning of the next line when necessary; a new line is also started if ENTER is pressed. The position of the wordwrap is determined by the window size. If the window size changes, the wordwrap position changes, and the text is redisplayed. Multiple-line edit controls can have scroll bars. An edit control with scroll bars processes its own scroll-bar messages. Edit controls without scroll bars scroll as described above, and process any scroll messages sent by the parent window.
ES_NOHIDESEL	Normally, an edit control hides the selection when the control loses the input focus, and inverts the selection when the control receives the input focus. Specifying ES_NOHIDESEL deletes this default action.
ES_OEMCONVERT	Text entered in the edit control is converted from the ANSI character set to the OEM character set and then back to ANSI. This ensures proper character conversion when the application calls the **AnsiToOem** function to convert an ANSI string in the edit control to OEM characters. This style is most useful for edit controls that contain filenames.
ES_PASSWORD	Displays all characters as an asterisk (*) as they are typed into the edit control. An application can use the EM_SETPASSWORDCHAR message to change the character that is displayed.
ES_RIGHT	Aligns text flush-right in a multiline edit control.
ES_UPPERCASE	Converts all characters to uppercase as they are typed into the edit control.
LISTBOX Class	
LBS_EXTENDEDSEL	The user can select multiple items using the SHIFT key and the mouse or special key combinations.

Table 4.4 Control Styles *(continued)*

Style	Meaning
LBS_HASSTRINGS	Specifies an owner-draw list box which contains items consisting of strings. The list box maintains the memory and pointers for the strings so the application can use the LB_GETTEXT message to retrieve the text for a particular item.
LBS_MULTICOLUMN	Specifies a multicolumn list box that is scrolled horizontally. The LB_SETCOLUMNWIDTH message sets the width of the columns.
LBS_MULTIPLESEL	String selection is toggled each time the user clicks or double-clicks the string. Any number of strings can be selected.
LBS_NOINTEGRALHEIGHT	The size of the list box is exactly the size specified by the application when it created the list box. Normally, Windows sizes a list box so that the list box does not display partial items.
LBS_NOREDRAW	List-box display is not updated when changes are made. This style can be changed at any time by sending a WM_SETREDRAW message.
LBS_NOTIFY	Parent window receives an input message whenever the user clicks or double-clicks a string.
LBS_OWNERDRAWFIXED	The owner of the list box is responsible for drawing its contents; the items in the list box are the same height.
LBS_OWNERDRAWVARIABLE	The owner of the list box is responsible for drawing its contents; the items in the list box are variable in height.
LBS_SORT	Strings in the list box are sorted alphabetically.
LBS_STANDARD	Strings in the list box are sorted alphabetically and the parent window receives an input message whenever the user clicks or double-clicks a string. The list box contains borders on all sides.
LBS_USETABSTOPS	Allows a list box to recognize and expand tab characters when drawing its strings. The default tab positions are 32 dialog units. (A dialog unit is a horizontal or vertical distance. One horizontal dialog unit is equal to 1/4 of the current dialog base width unit. The dialog base units are computed based on the height and width of the current system font. The **GetDialogBaseUnits** function returns the current dialog base units in pixels.)

Table 4.4 Control Styles (*continued*)

Style	Meaning
LBS_WANTKEYBOARDINPUT	The owner of the list box receives WM_VKEY-TOITEM or WM_CHARTOITEM messages whenever the user presses a key when the list box has input focus. This allows an application to perform special processing on the keyboard input.
SCROLLBAR Class	
SBS_BOTTOMALIGN	Used with the SBS_HORZ style. The bottom edge of the scroll bar is aligned with the bottom edge of the rectangle specified by the *X*, *Y*, *nWidth*, and *nHeight* parameters given in the **CreateWindow** function. The scroll bar has the default height for system scroll bars.
SBS_HORZ	Designates a horizontal scroll bar. If neither the SBS_BOTTOMALIGN nor SBS_TOPALIGN style is specified, the scroll bar has the height, width, and position given in the **CreateWindow** function.
SBS_LEFTALIGN	Used with the SBS_VERT style. The left edge of the scroll bar is aligned with the left edge of the rectangle specified by the *X*, *Y*, *nWidth*, and *nHeight* parameters given in the **CreateWindow** function. The scroll bar has the default width for system scroll bars.
SBS_RIGHTALIGN	Used with the SBS_VERT style. The right edge of the scroll bar is aligned with the right edge of the rectangle specified by the *X*, *Y*, *nWidth*, and *nHeight* parameters given in the **CreateWindow** function. The scroll bar has the default width for system scroll bars.
SBS_SIZEBOX	Designates a size box. If neither the SBS_SIZE-BOXBOTTOMRIGHTALIGN nor SBS_SIZEBOXTOPLEFTALIGN style is specified, the size box has the height, width, and position given in the **CreateWindow** function.
SBS_SIZEBOXBOTTOMRIGHTALIGN	Used with the SBS_SIZEBOX style. The lower-right corner of the size box is aligned with the lower-right corner of the rectangle specified by the *X*, *Y*, *nWidth*, and *nHeight* parameters given in the **CreateWindow** function. The size box has the default size for system size boxes.

Table 4.4 Control Styles (*continued*)

Style	Meaning
SBS_SIZEBOXTOPLEFTALIGN	Used with the SBS_SIZEBOX style. The upper-left corner of the size box is aligned with the upper-left corner of the rectangle specified by the *X*, *Y*, *nWidth*, and *nHeight* parameters given in the **CreateWindow** function. The size box has the default size for system size boxes.
SBS_TOPALIGN	Used with the SBS_HORZ style. The top edge of the scroll bar is aligned with the top edge of the rectangle specified by the *X*, *Y*, *nWidth*, and *nHeight* parameters given in the **CreateWindow** function. The scroll bar has the default height for system scroll bars.
SBS_VERT	Designates a vertical scroll bar. If neither the SBS_RIGHTALIGN nor SBS_LEFTALIGN style is specified, the scroll bar has the height, width, and position given in the **CreateWindow** function.
STATIC Class	
SS_BLACKFRAME	Specifies a box with a frame drawn with the same color as window frames. This color is black in the default Windows color scheme.
SS_BLACKRECT	Specifies a rectangle filled with the color used to draw window frames. This color is black in the default Windows color scheme.
SS_CENTER	Designates a simple rectangle and displays the given text centered in the rectangle. The text is formatted before it is displayed. Words that would extend past the end of a line are automatically wrapped to the beginning of the next centered line.
SS_GRAYFRAME	Specifies a box with a frame drawn with the same color as the screen background (desktop). This color is gray in the default Windows color scheme.
SS_GRAYRECT	Specifies a rectangle filled with the color used to fill the screen background. This color is gray in the default Windows color scheme.
SS_ICON	Designates an icon displayed in the dialog box. The given text is the name of an icon (not a filename) defined elsewhere in the resource file. The *nWidth* and *nHeight* parameters are ignored; the icon automatically sizes itself.

Table 4.4 Control Styles (*continued*)

Style	Meaning
SS_LEFT	Designates a simple rectangle and displays the given text flush-left in the rectangle. The text is formatted before it is displayed. Words that would extend past the end of a line are automatically wrapped to the beginning of the next flush-left line.
SS_LEFTNOWORDWRAP	Designates a simple rectangle and displays the given text flush-left in the rectangle. Tabs are expanded, but words are not wrapped. Text that extends past the end of a line is clipped.
SS_NOPREFIX	Unless this style is specified, windows will interpret any "&" characters in the control's text to be accelerator prefix characters. In this case, the "&" is removed and the next character in the string is underlined. If a static control is to contain text where this feature is not wanted, SS_NOPREFIX may be added. This static-control style may be included with any of the defined static controls. You can combine SS_NOPREFIX with other styles by using the bitwise OR operator. This is most often used when filenames or other strings that may contain an "&" need to be displayed in a static control in a dialog box.
SS_RIGHT	Designates a simple rectangle and displays the given text flush-right in the rectangle. The text is formatted before it is displayed. Words that would extend past the end of a line are automatically wrapped to the beginning of the next flush-right line.
SS_SIMPLE	Designates a simple rectangle and displays a single line of text flush-left in the rectangle. The line of text cannot be shortened or altered in any way. (The control's parent window or dialog box must not process the WM_CTLCOLOR message.)
SS_USERITEM	Specifies a user-defined item.
SS_WHITEFRAME	Specifies a box with a frame drawn with the same color as window backgrounds. This color is white in the default Windows color scheme.
SS_WHITERECT	Specifies a rectangle filled with the color used to fill window backgrounds. This color is white in the default Windows color scheme.

CreateWindowEx 3.0

Syntax **HWND CreateWindowEx**(*dwExStyle*, *lpClassName*, *lpWindowName*, *dwStyle*, *X*, *Y*, *nWidth*, *nHeight*, *hWndParent*, *hMenu*, *hInstance*, *lpParam*)

This function creates an overlapped, pop-up, or child window with an extended style specified in the *dwExStyle* parameter. Otherwise, this function is identical to the **CreateWindow** function. See the description of the **CreateWindow** function for more information on creating a window and for a full descriptions of the other parameters of **CreateWindowEx**.

Parameter	Type/Description
dwExStyle	**DWORD** Specifies the extended style of the window being created. Table 4.5, "Extended Window Styles," lists the extended window styles.
lpClassName	**LPSTR** Points to a null-terminated character string that names the window class.
lpWindowName	**LPSTR** Points to a null-terminated character string that represents the window name.
dwStyle	**DWORD** Specifies the style of window being created.
X	**int** Specifies the initial *x*-position of the window.
Y	**int** Specifies the initial *y*-position of the window.
nWidth	**int** Specifies the width (in device units) of the window.
nHeight	**int** Specifies the height (in device units) of the window.
hWndParent	**HWND** Identifies the parent or owner window of the window being created.
hMenu	**HMENU** Identifies a menu or a child-window identifier. The meaning depends on the window style.
hInstance	**HANDLE** Identifies the instance of the module to be associated with the window.
lpParam	**LPSTR** Points to a value that is passed to the window through the **CREATESTRUCT** data structure referenced by the *lParam* parameter of the WM_CREATE message.

Return Value The return value identifies the new window. It is NULL if the window is not created.

Comments Table 4.5 lists the extended window styles.

Table 4.5 Extended Window Styles

Style	Meaning
WS_EX_DLGMODALFRAME	Designates a window with a double border that may optionally be created with a title bar by specifying the WS_CAPTION style flag in the *dwStyle* parameter.
WS_EX_NOPARENTNOTIFY	Specifies that a child window created with this style will not send the WM_PARENTNOTIFY message to its parent window when the child window is created or destroyed.

Table 4.2, "Control Classes," lists the window control classes. Table 4.3, "Window Styles," lists the window styles. Table 4.4, "Control Styles," lists the control styles. See the description of the **CreateWindow** function for these tables.

DebugBreak 3.0

Syntax **void DebugBreak( )**

This function forces a break to the debugger.

This function has no parameters.

Return Value None.

DefDlgProc 3.0

Syntax **LONG DefDlgProc(***hDlg*, *wMsg*, *wParam*, *lParam*)

This function provides default processing for any Windows messages that a dialog box with a private window class does not process.

All window messages that are not explicitly processed by the window function must be passed to the **DefDlgProc** function, not the **DefWindowProc** function. This ensures that all messages not handled by their private window procedure will be handled properly.

Parameter	Type/Description
hDlg	**HWND** Identifies the dialog box.
wMsg	**WORD** Specifies the message number.
wParam	**WORD** Specifies 16 bits of additional message-dependent information.
lParam	**DWORD** Specifies 32 bits of additional message-dependent information.

Return Value The return value specifies the result of the message processing and depends on the actual message sent.

Comments The source code for the **DefDlgProc** function is provided on the SDK disks.

An application creates a dialog box by calling one of the following functions:

Function	Description
CreateDialog	Creates a modeless dialog box.
CreateDialogIndirect	Creates a modeless dialog box.

Function	Description
CreateDialogIndirectParam	Creates a modeless dialog box and passes data to it when it is created.
CreateDialogParam	Creates a modeless dialog box and passes data to it when it is created.
DialogBox	Creates a modal dialog box.
DialogBoxIndirect	Creates a modal dialog box.
DialogBoxIndirectParam	Creates a modal dialog box and passes data to it when it is created.
DialogBoxParam	Creates a modal dialog box and passes data to it when it is created.

DeferWindowPos 3.0

Syntax HANDLE **DeferWindowPos**(*hWinPosInfo*, *hWnd*, *hWndInsertAfter*, *x*, *y*, *cx*, *cy*, *wFlags*)

This function updates the multiple window-position data structure identified by the *hWinPosInfo* parameter for the window identified by *hWnd* parameter and returns the handle of the updated structure. The **EndDeferWindowPos** function uses the information in this structure to change the position and size of a number of windows simultaneously. The **BeginDeferWindowPos** function creates the multiple window-position data structure used by this function.

The *x* and *y* parameters specify the new position of the window, and the *cx* and *cy* parameters specify the new size of the window.

Parameter	Type/Description
hWinPosInfo	**HANDLE** Identifies a multiple window-position data structure that contains size and position information for one or more windows. This structure is returned by the **BeginDeferWindowPos** function or the most recent call to the **DeferWindowPos** function.
hWnd	**HWND** Identifies the window for which update information is to be stored in the data structure.
hWndInsertAfter	**HWND** Identifies the window following which the window identified by the *hWnd* parameter is to be updated.

Parameter	Type/Description
x	**int** Specifies the *x*-coordinate of the window's upper-left corner.
y	**int** Specifies the *y*-coordinate of the window's upper-left corner.
cx	**int** Specifies the window's new width.
cy	**int** Specifies the window's new height.
wFlags	**WORD** Specifies one of eight possible 16-bit values that affect the size and position of the window. It must be one of the following values:

Value	Meaning
SWP_DRAWFRAME	Draws a frame (defined in the window's class description) around the window.
SWP_HIDEWINDOW	Hides the window.
SWP_NOACTIVATE	Does not activate the window.
SWP_NOMOVE	Retains current position (ignores the *x* and *y* parameters).
SWP_NOREDRAW	Does not redraw changes.
SWP_NOSIZE	Retains current size (ignores the *cx* and *cy* parameters).
SWP_NOZORDER	Retains current ordering (ignores the *hWndInsertAfter* parameter).
SWP_SHOWWINDOW	Displays the window.

Return Value

The return value identifies the updated multiple window-position data structure. The handle returned by this function may differ from the handle passed to the function as the *hWinPosInfo* parameter. The new handle returned by this function should be passed to the next call to **DeferWindowPos** or the **EndDeferWindowPos** function.

The return value is NULL if insufficient system resources are available for the function to complete successfully.

Comments If the SWP_NOZORDER flag is not specified, Windows places the window identified by the *hWnd* parameter in the position following the window identified by the *hWndInsertAfter* parameter. If *hWndInsertAfter* is NULL, Windows places the window identified by *hWnd* at the top of the list. If *hWndInsertAfter* is set to 1, Windows places the window identified by *hWnd* at the bottom of the list.

If the SWP_SHOWWINDOW or the SWP_HIDEWINDOW flags are set, scrolling and moving cannot be done simultaneously.

All coordinates for child windows are relative to the upper-left corner of the parent window's client area.

DefFrameProc 3.0

Syntax **LONG DefFrameProc**(*hWnd*, *hWndMDIClient*, *wMsg*, *wParam*, *lParam*)

This function provides default processing for any Windows messages that the window function of a multiple document interface (MDI) frame window does not process. All window messages that are not explicitly processed by the window function must be passed to the **DefFrameProc** function, not the **DefWindowProc** function.

Parameter	Type/Description
hWnd	**HWND** Identifies the MDI frame window.
hWndMDIClient	**HWND** Identifies the MDI client window.
wMsg	**WORD** Specifies the message number.
wParam	**WORD** Specifies 16 bits of additional message-dependent information.
lParam	**DWORD** Specifies 32 bits of additional message-dependent information.

Return Value The return value specifies the result of the message processing and depends on the actual message sent. If the *hWndMDIClient* parameter is NULL, the return value is the same as for the **DefWindowProc** function.

Comments Normally, when an application's window procedure does not handle a message, it passes the message to the **DefWindowProc** function, which processes the message. MDI applications use the **DefFrameProc** and **DefMDIChildProc** functions instead of **DefWindowProc** to provide default message processing. All messages that an application would normally pass to **DefWindowProc** (such as nonclient messages and WM_SETTEXT) should be passed to **DefFrameProc** instead. In addition to these, **DefFrameProc** also handles the following messages:

Message	Default Processing by DefFrameProc
WM_COMMAND	The frame window of an MDI application receives the WM_COMMAND message to activate a particular MDI child window. The window ID accompanying this message will be the ID of the MDI child window assigned by Windows, starting with the first ID specified by the application when it created the MDI client window. This value of the first ID must not conflict with menu-item IDs.
WM_MENUCHAR	When the ALT+HYPHEN key is pressed, the control menu of the active MDI child window will be selected.
WM_NEXTMENU	This message causes the control menu of the active MDI child window to be selected.
WM_SETFOCUS	**DefFrameProc** passes focus on to the MDI client, which in turn passes the focus on to the active MDI child window.
WM_SIZE	If the frame window procedure passes this message to **DefFrameProc**, the MDI client window will be resized to fit in the new client area. If the frame window procedure sizes the MDI client to a different size, it should not pass the message to **DefWindowProc**.

DefHookProc

Syntax **DWORD DefHookProc**(*code*, *wParam*, *lParam*, *lplpfnNextHook*)

This function calls the next function in a chain of hook functions. A hook function is a function that processes events before they are sent to an application's message-processing loop in the WinMain function. When an application defines more than one hook function by using the **SetWindowsHook** function, Windows forms a linked list or hook chain. Windows places functions of the same type in a chain.

Parameter	Type/Description
code	**int** Specifies a code used by the Windows hook function (also called the message filter function) to determine how to process the message.
wParam	**WORD** Specifies the word parameter of the message that the hook function is processing.
lParam	**DWORD** Specifies the long parameter of the message that the hook function is processing.

Parameter	Type/Description
lplpfnNextHook	**FARPROC FAR *** Points to a memory location that contains the **FARPROC** structure returned by the **SetWindowsHook** function. Windows changes the value at this location after an application calls the **UnhookWindowsHook** function.

Return Value The return value specifies a value that is directly related to the *code* parameter.

DefineHandleTable 3.0

Syntax **BOOL DefineHandleTable**(*wOffset*)

This function creates a private handle table in an application's default data segment. The application stores in the table the segment addresses of global memory objects returned by the **GlobalLock** function. In real mode, Windows updates the corresponding address in the private handle table when it moves a global memory object. When Windows discards an object with a corresponding table entry, Windows replaces the address of the object in the table with the object's handle. Windows does not update addresses in the private handle table in protected (standard or 386 enhanced) mode.

Parameter	Type/Description
wOffset	**WORD** Specifies the offset from the beginning of the data segment to the beginning of the private handle table. If *wOffset* is zero, Windows no longer updates the private handle table.

Return Value The return value is nonzero if the function was successful. Otherwise, it is zero.

Comments The private handle table has the following format:

```
Count
Clear_Number
Entry[0]
.
.
.
Entry[Count-1]
```

The first **WORD** (*Count*) in the table specifies the number of entries in the table. The second **WORD** (*Clear_Number*) specifies the number of entries (from the beginning of the table) which Windows will set to zero when Windows updates its least-recently-used

(LRU) memory list. The remainder of the table consists of an array of addresses returned by **GlobalLock**.

The application must initialize the *Count* field in the table before calling **DefineHandleTable**. The application can change either the *Count* or *Clearn_Number* fields at any time.

DefMDIChildProc 3.0

Syntax LONG **DefMDIChildProc**(*hWnd*, *wMsg*, *wParam*, *lParam*)

This function provides default processing for any Windows messages that the window function of a multiple document interface (MDI) child window does not process. All window messages that are not explicitly processed by the window function must be passed to the **DefMDIChildProc** function, not the **DefWindowProc** function.

Parameter	Type/Description
hWnd	**HWND** Identifies the MDI child window.
wMsg	**WORD** Specifies the message number.
wParam	**WORD** Specifies 16 bits of additional message-dependent information.
lParam	**DWORD** Specifies 32 bits of additional message-dependent information.

Return Value The return value specifies the result of the message processing and depends on the actual message sent.

Comments This function assumes that the parent of the window identified by the *hWnd* parameter was created with the MDICLIENT class.

Normally, when an application's window procedure does not handle a message, it passes the message to the **DefWindowProc** function, which processes the message. MDI applications use the **DefFrameProc** and **DefMDIChildProc** functions instead of **DefWindowProc** to provide default message processing. All messages that an application would normally pass to **DefWindowProc** (such as nonclient messages and WM_SETTEXT) should be passed to **DefMDIChildProc** instead. In addition to these, **DefMDIChildProc** also handles the following messages:

Message	Default Processing by DefMDIChildProc
WM_CHILDACTIVATE	Performs activation processing when child windows are sized, moved, or shown. This message must be passed.
WM_GETMINMAXINFO	Calculates the size of a maximized MDI child window based on the current size of the MDI client window.
WM_MENUCHAR	Sends the key to the frame window.
WM_MOVE	Recalculates MDI client scroll bars, if they are present.
WM_NEXTMENU	Wraps back to the frame menu bar or frame control menu.
WM_SETFOCUS	Activates the child window if it is not the active MDI child.
WM_SIZE	Performs necessary operations when changing the size of a window, especially when maximizing or restoring an MDI child window. Failing to pass this message to **DefMDIChildProc** will produce highly undesirable results.
WM_SYSCOMMAND	Also handles the "next window" command.

DefWindowProc

Syntax LONG **DefWindowProc**(*hWnd*, *wMsg*, *wParam*, *lParam*)

This function provides default processing for any Windows messages that a given application does not process. All window messages that are not explicitly processed by the class window function must be passed to the **DefWindowProc** function.

Parameter	Type/Description
hWnd	**HWND** Identifies the window that passes the message.
wMsg	**WORD** Specifies the message number.
wParam	**WORD** Specifies 16 bits of additional message-dependent information.

Parameter	Type/Description
lParam	**DWORD** Specifies 32 bits of additional message-dependent information.

Return Value The return value specifies the result of the message processing and depends on the actual message sent.

Comments The source code for the **DefWindowProc** function is provided on the SDK disks.

DeleteAtom

Syntax ATOM **DeleteAtom**(*nAtom*)

This function deletes an atom and, if the atom's reference count is zero, removes the associated string from the atom table.

An atom's reference count specifies the number of times the atom has been added to the atom table. The **AddAtom** function increases the count on each call; the **DeleteAtom** function decreases the count on each call. **DeleteAtom** removes the string only if the atom's reference count is zero.

Parameter	Type/Description
nAtom	**ATOM** Identifies the atom and character string to be deleted.

Return Value The return value specifies the outcome of the function. It is NULL if the function is successful. It is equal to the *nAtom* parameter if the function failed and the atom has not been deleted.

DeleteDC

Syntax BOOL **DeleteDC**(*hDC*)

This function deletes the specified device context. If the *hDC* parameter is the last device context for a given device, the device is notified and all storage and system resources used by the device are released.

Parameter	Type/Description
hDC	**HDC** Identifies the device context.

Return Value The return value specifies whether the device context is deleted. It is nonzero if the device context is successfully deleted (regardless of whether the deleted device context is the last context for the device). If an error occurs, the return value is zero.

Comments An application must not delete a device context whose handle was obtained by calling the **GetDC** function. Instead, it must call the **ReleaseDC** function to free the device context.

DeleteMenu 3.0

Syntax BOOL **DeleteMenu**(*hMenu*, *nPosition*, *wFlags*)

This function deletes an item from the menu identified by the *hMenu* parameter; if the menu item has an associated pop-up menu, **DeleteMenu** destroys the handle by the pop-up menu and frees the memory used by the pop-up menu.

Parameter	Type/Description
hMenu	**HMENU** Identifies the menu to be changed.
nPosition	**WORD** Specifies the menu item which is to be deleted. If *wFlags* is set to MF_BYPOSITION, *nPosition* specifies the position of the menu item; the first item in the menu is at position 0. If *wFlags* is set to MF_BYCOMMAND, then *nPosition* specifies the command ID of the existing menu item.
wFlags	**WORD** Specifies how the *nPosition* parameter is interpreted. It may be set to either MF_BYCOMMAND (the default) or MF_BYPOSITION.

Return Value The return value specifies the outcome of the function. It is TRUE if the function is successful. Otherwise, it is FALSE.

Comments Whenever a menu changes (whether or not the menu resides in a window that is displayed), the application should call **DrawMenuBar**.

DeleteMetaFile

Syntax BOOL **DeleteMetaFile**(*hMF*)

This function deletes access to a metafile by freeing the system resources associated with that metafile. It does not destroy the metafile itself, but it invalidates the metafile handle, *hMF*. Access to the metafile can be reestablished by retrieving a new handle by using the **GetMetaFile** function.

Parameter	Type/Description
hMF	**HANDLE** Identifies the metafile to be deleted.

Return Value

The return value specifies whether the metafile handle is invalidated. It is nonzero if the metafile's system resources are deleted. It is zero if the *hMF* parameter is not a valid handle.

DeleteObject

Syntax

BOOL DeleteObject(*hObject*)

This function deletes a logical pen, brush, font, bitmap, region, or palette from memory by freeing all system storage associated with the object. After the object is deleted, the *hObject* handle is no longer valid.

Parameter	Type/Description
hObject	**HANDLE** Identifies a handle to a logical pen, brush, font, bitmap, region, or palette.

Return Value

The return value specifies whether the specified object is deleted. It is nonzero if the object is deleted. It is zero if the *hObject* parameter is not a valid handle or is currently selected into a device context.

Comments

The object to be deleted should not be currently selected into a device context.

When a pattern brush is deleted, the bitmap associated with the brush is not deleted. The bitmap must be deleted independently.

An application must not delete a stock object.

DestroyCaret

Syntax

void DestroyCaret( )

This function destroys the current caret shape, frees the caret from the window that currently owns it, and removes the caret from the screen if it is visible. The **DestroyCaret** function checks the ownership of the caret and destroys the caret only if a window in the current task owns it.

If the caret shape was previously a bitmap, **DestroyCaret** does not free the bitmap.

This function has no parameters.

Return Value None.

Comments The caret is a shared resource. If a window has created a caret shape, it destroys that shape before it loses the input focus or becomes inactive.

DestroyCursor 3.0

Syntax **BOOL DestroyCursor**(*hCursor*)

This function destroys a cursor that was previously created by the **CreateCursor** function and frees any memory that the cursor occupied. It should not be used to destroy any cursor that was not created with the **CreateCursor** function.

Parameter	Type/Description
hCursor	**HCURSOR** Identifies the cursor to be destroyed. The cursor must not be in current use.

Return Value The return value is nonzero if the function was successful. It is zero if the function failed.

DestroyIcon 3.0

Syntax **BOOL DestroyIcon**(*hIcon*)

This function destroys an icon that was previously created by the **CreateIcon** function and frees any memory that the icon occupied. It should not be used to destroy any icon that was not created with the **CreateIcon** function.

Parameter	Type/Description
hIcon	**HICON** Identifies the icon to be destroyed. The icon must not be in current use.

Return Value The return value is nonzero if the function was successful. It is zero if the function failed.

DestroyMenu

Syntax BOOL **DestroyMenu**(*hMenu*)

This function destroys the menu specified by the *hMenu* parameter and frees any memory that the menu occupied.

Parameter	Type/Description
hMenu	**HMENU** Identifies the menu to be destroyed.

Return Value The return value specifies whether or not the specified menu is destroyed. It is nonzero if the menu is destroyed. Otherwise, it is zero.

DestroyWindow

Syntax BOOL **DestroyWindow**(*hWnd*)

This function destroys the specified window. The **DestroyWindow** function hides or permanently closes the window, sending the appropriate messages to the window to deactivate it and remove the input focus. It also destroys the window menu, flushes the application queue, destroys outstanding timers, removes clipboard ownership, and breaks the clipboard-viewer chain, if the window is at the top of the viewer chain. It sends WM_DESTROY and WM_NCDESTROY messages to the window.

If the given window is the parent of any windows, these child windows are automatically destroyed when the parent window is destroyed. **DestroyWindow** destroys child windows first, and then the window itself.

DestroyWindow also destroys modeless dialog boxes created by the **CreateDialog** function.

Parameter	Type/Description
hWnd	**HWND** Identifies the window to be destroyed.

Return Value The return value specifies whether or not the specified window is destroyed. It is nonzero if the window is destroyed. Otherwise, it is zero.

DeviceCapabilities 3.0

Syntax **DWORD DeviceCapabilities**(*lpDeviceName*, *lpPort*, *nIndex*, *lpOutput*, *lpDevMode*)

This function retrieves the capabilities of the printer device driver.

Parameter	Type/Description
lpDeviceName	**LPSTR** Points to a null-terminated character string that contains the name of the printer device, such as "PCL/HP LaserJet."
lpPort	**LPSTR** Points to a null-terminated character string that contains the name of the port to which the device is connected, such as LPT1:.
nIndex	**WORD** Specifies the capabilities to query. It can be any one of the following values:

Value	Meaning
DC_BINNAMES	Copies a structure identical to that returned by the **ENUMPAPERBINS** escape. A printer driver does not need to support this index if it has only bins corresponding to predefined indexes, in which case no data is copied and the return value is 0. If the index is supported, the return value is the number of bins copied. If *lpOutput* is NULL, the return value is the number of bin entries required.
DC_BINS	Retrieves a list of available bins. The function copies the list to *lpOutput* as a **WORD** array. If *lpOutput* is NULL, the function returns the number of supported bins to allow the application the opportunity to allocate a buffer with the correct size. See the description of the **dmDefaultSource** field of the **DEVMODE** data structure for information on these values. An application can determine the name of device-specific bins by using the **ENUMPAPERBINS** escape.

D

Parameter	Type/Description	
	Value	**Meaning**
	DC_DRIVER	Returns the printer driver version number.
	DC_DUPLEX	Returns the level of duplex support. The function returns 1 if the printer is capable of duplex printing. Otherwise, the return value is zero.
	DC_EXTRA	Returns the number of bytes required for the device-specific portion of the **DEVMODE** data structure for the printer driver.
	DC_FIELDS	Returns the **dmFields** field of the printer driver's **DEVMODE** data structure. The **dmFields** bitfield indicates which fields in the device-independent portion of the structure are supported by the printer driver.
	DC_MAXEXTENT	Returns a **POINT** data structure containing the maximum paper size that the **dmPaperLength** and **dmPaperWidth** fields of the printer driver's **DEVMODE** data structure can specify.
	DC_MINEXTENT	Returns a **POINT** data structure containing the minimum paper size that the **dmPaperLength** and **dmPaperWidth** fields of the printer driver's **DEVMODE** data structure can specify.

Parameter	Type/Description	
	Value	**Meaning**
	DC_PAPERS	Retrieves a list of supported paper sizes. The function copies the list to *lpOutput* as a **WORD** array and returns the number of entries in the array. If *lpOutput* is NULL, the function returns the number of supported paper sizes to allow the application the opportunity to allocate a buffer with the correct size. See the description of the **dmPaperSize** field of the **DEVMODE** data structure for information on these values.
	DC_PAPERSIZE	Copies the dimensions of supported paper sizes in tenths of a millimeter to an array of **POINT** structures in *lpOutput*. This allows an application to obtain information about nonstandard paper sizes.
	DC_SIZE	Returns the **dmSize** field of the printer driver's **DEVMODE** data structure.
	DC_VERSION	Returns the specification version to which the printer driver conforms.
lpOutput	**LPSTR** Points to an array of bytes. The actual format of the array depends on the setting of *nIndex*. If set to zero, **DeviceCapabilities** returns the number of bytes required for the output data.	
lpDevMode	**DEVMODE FAR *** Points to a **DEVMODE** data structure. If *lpDevMode* is NULL, this function retrieves the current default initialization values for the specified printer driver. Otherwise, the function retrieves the values contained in the structure to which *lpDevMode* points.	

Return Value

The return value depends on the setting of the *nIndex* parameter; see the description of that parameter for details.

Comments This function is supplied by the printer driver. An application must include the DRIVINIT.H file and call the **LoadLibrary** and **GetProcAddress** functions to call the **DeviceCapabilities** function.

DeviceMode

Syntax **void DeviceMode**(*hWnd*, *hModule*, *lpDeviceName*, *lpOutput*)

This function sets the current printing modes for the device identified by the *lpDestDevType* by prompting for those modes using a dialog box. An application calls the **DeviceMode** function to allow the user to change the printing modes of the corresponding device. The function copies the mode information to the environment block associated with the device and maintained by GDI.

Parameter	Type/Description
hWnd	**HWND** Identifies the window that will own the dialog box.
hModule	**HANDLE** Identifies the printer-driver module. The application should retrieve this handle by calling either the **GetModuleHandle** or **LoadLibrary** function.
lpDeviceName	**LPSTR** Points to a null-terminated character string that specifies the name of the specific device to be supported (for example, Epson FX-80). The device name is the same as the name passed to the **CreateDC** function.
lpOutput	**LPSTR** Points to a null-terminated character string that specifies the DOS file or device name for the physical output medium (file or output port). The output name is the same as the name passed to the **CreateDC** function.

Return Value None.

Comments The **DeviceMode** function is actually part of the printer's device driver, and not part of GDI. To call this function, the application must load the printer device driver by calling **LoadLibrary** and retrieve the address of the function by using the **GetProcAddress** function. The application can then use the address to set up the printer.

DialogBox

Syntax

int DialogBox(*hInstance*, *lpTemplateName*, *hWndParent*, *lpDialogFunc*)

This function creates a modal dialog box that has the size, style, and controls specified by the dialog-box template given by the *lpTemplateName* parameter. The *hWndParent* parameter identifies the application window that owns the dialog box. The callback function pointed to by the *lpDialogFunc* parameter processes any messages received by the dialog box.

The **DialogBox** function does not return control until the callback function terminates the modal dialog box by calling the **EndDialog** function.

Parameter	Type/Description
hInstance	**HANDLE** Identifies an instance of the module whose executable file contains the dialog-box template.
lpTemplateName	**LPSTR** Points to a character string that names the dialog-box template. The string must be a null-terminated character string.
hWndParent	**HWND** Identifies the window that owns the dialog box.
lpDialogFunc	**FARPROC** Is the procedure-instance address of the dialog function. See the following "Comments" section for details.

Return Value

The return value specifies the value of the *nResult* parameter in the **EndDialog** function that is used to terminate the dialog box. Values returned by the application's dialog box are processed by Windows and are not returned to the application. The return value is –1 if the function could not create the dialog box.

Comments

The **DialogBox** function calls the **GetDC** function in order to obtain a display-context. Problems will result if all the display contexts in the Windows display-context cache have been retrieved by **GetDC** and **DialogBox** attempts to access another display context.

A dialog box can contain up to 255 controls.

The callback function must use the Pascal calling convention and must be declared **FAR**.

Callback Function

int FAR PASCAL *DialogFunc*(*hDlg*, *wMsg*, *wParam*, *lParam*)
HWND *hDlg*;
WORD *wMsg*;
WORD *wParam*;
DWORD *lParam*;

DialogFunc is a placeholder for the application-supplied function name. The actual name must be exported by including it in an **EXPORTS** statement in the application's module-definition file.

Parameter	Description
hDlg	Identifies the dialog box that receives the message.
wMsg	Specifies the message number.
wParam	Specifies 16 bits of additional message-dependent information.
lParam	Specifies 32 bits of additional message-dependent information.

Return Value

The callback function should return nonzero if the function processes the message and zero if it does not.

Comments

Although the callback function is similar to a window function, it must not call the **DefWindowProc** function to process unwanted messages. Unwanted messages are processed internally by the dialog-class window function.

The callback-function address, passed as the *lpDialogFunc* parameter, must be created by using the **MakeProcInstance** function.

DialogBoxIndirect

Syntax **int DialogBoxIndirect**(*hInstance*, *hDialogTemplate*, *hWndParent*, *lpDialogFunc*)

This function creates an application's modal dialog box that has the size, style, and controls specified by the dialog-box template associated with the *hDialogTemplate* parameter. The *hWndParent* parameter identifies the application window that owns the dialog box. The callback function pointed to by *lpDialogFunc* processes any messages received by the dialog box.

The **DialogBoxIndirect** function does not return control until the callback function terminates the modal dialog box by calling the **EndDialog** function.

Parameter	Type/Description
hInstance	**HANDLE** Identifies an instance of the module whose executable file contains the dialog-box template.

Parameter	Type/Description
hDialogTemplate	**HANDLE** Identifies a block of global memory that contains a **DLGTEMPLATE** data structure.
hWndParent	**HWND** Identifies the window that owns the dialog box.
lpDialogFunc	**FARPROC** Is the procedure-instance address of the dialog function. See the following "Comments" section for details.

Return Value

The return value specifies the value of the *wResult* parameter specified in the **EndDialog** function that is used to terminate the dialog box. Values returned by the application's dialog box are processed by Windows and are not returned to the application. The return value is –1 if the function could not create the dialog box.

Comments

A dialog box can contain up to 255 controls.

The callback function must use the Pascal calling convention and be declared **FAR**.

Callback Function

BOOL FAR PASCAL *DialogFunc(hDlg, wMsg, wParam, lParam)*
HWND *hDlg*;
WORD *wMsg*;
WORD *wParam*;
DWORD *lParam*;

DialogFunc is a placeholder for the application-supplied function name. The actual name must be exported by including it in an **EXPORTS** statement in the application's module-definition file.

Parameter	Description
hDlg	Identifies the dialog box that receives the message.
wMsg	Specifies the message number.
wParam	Specifies 16 bits of additional message-dependent information.
lParam	Specifies 32 bits of additional message-dependent information.

Return Value

The callback function should return nonzero if the function processes the message and zero if it does not.

Comments

Although the callback function is similar to a window function, it must not call the **Def-WindowProc** function to process unwanted messages. Unwanted messages are processed internally by the dialog-class window function.

The callback-function address, passed as the *lpDialogFunc* parameter, must be created by using the **MakeProcInstance** function.

DialogBoxIndirectParam 3.0

D

Syntax

int DialogBoxIndirectParam(*hInstance*, *hDialogTemplate*, *hWndParent*, *lpDialogFunc*, *dwInitParam*)

This function creates an application's modal dialog box, sends a WM_INITDIALOG message to the dialog function before displaying the dialog box and passes *dwInitParam* as the message *lParam*. This message allows the dialog function to initialize the dialog-box controls.

For more information on creating an application modal dialog box, see the description of the **DialogBoxIndirect** function.

Parameter	Type/Description
hInstance	**HANDLE** Identifies an instance of the module whose executable file contains the dialog-box template.
hDialogTemplate	**HANDLE** Identifies a block of global memory that contains a **DLGTEMPLATE** data structure.
hWndParent	**HWND** Identifies the window that owns the dialog box.
lpDialogFunc	**FARPROC** Is the procedure-instance address of the dialog function. For details, see the "Comments" section in the description of the **DialogBoxIndirect** function.
dwInitParam	**DWORD** Is a 32-bit value which **DialogBoxIndirectParam** passes to the dialog function when it creates the dialog box.

Return Value

The return value specifies the value of the *wResult* parameter specified in the **EndDialog** function that is used to terminate the dialog box. Values returned by the application's dialog box are processed by Windows and are not returned to the application. The return value is –1 if the function could not create the dialog box.

DialogBoxParam 3.0

Syntax **int DialogBoxParam**(*hInstance, lpTemplateName, hWndParent, lpDialogFunc, dwInitParam*)

This function creates a modal dialog box, sends a WM_INITDIALOG message to the dialog function before displaying the dialog box, and passes *dwInitParam* as the message *lParam*. This message allows the dialog function to initialize the dialog-box controls.

For more information on creating a modal dialog box, see the description of the **DialogBox** function.

Parameter	Type/Description
hInstance	**HANDLE** Identifies an instance of the module whose executable file contains the dialog-box template.
lpTemplateName	**LPSTR** Points to a character string that names the dialog-box template. The string must be a null-terminated character string.
hWndParent	**HWND** Identifies the window that owns the dialog box.
lpDialogFunc	**FARPROC** Is the procedure-instance address of the dialog function. For details, see the "Comments" section of the description of the **DialogBox** function.
dwInitParam	**DWORD** Is a 32-bit value which **DialogBoxParam** passes to the dialog function when it creates the dialog box.

Return Value The return value specifies the value of the *nResult* parameter in the **EndDialog** function that is used to terminate the dialog box. Values returned by the application's dialog box are processed by Windows and are not returned to the application. The return value is –1 if the function could not create the dialog box.

DispatchMessage

Syntax **LONG DispatchMessage**(*lpMsg*)

This function passes the message in the **MSG** structure pointed to by the *lpMsg* parameter to the window function of the specified window.

Parameter	Type/Description
lpMsg	**LPMSG** Points to an **MSG** data structure that contains message information from the Windows application queue. The structure must contain valid message values. If *lpMsg* points to a WM_TIMER message and the *lParam* parameter of the WM_TIMER message is not NULL, then the *lParam* parameter is the address of a function that is called instead of the window function.

Return Value The return value specifies the value returned by the window function. Its meaning depends on the message being dispatched, but generally the return value is ignored.

DlgDirList

Syntax **int DlgDirList**(*hDlg, lpPathSpec, nIDListBox, nIDStaticPath, wFiletype*)

This function fills a list-box control with a file or directory listing. It fills the list box specified by the *nIDListBox* parameter with the names of all files matching the pathname given by the *lpPathSpec* parameter.

The **DlgDirList** function shows subdirectories enclosed in square brackets ([ ]), and shows drives in the form [–*x*–], where *x* is the drive letter.

The *lpPathSpec* parameter has the following form:

⟦*drive*:⟧ ⟦ ⟦\⟧*directory*⟦*directory*⟧...\⟧ ⟦*filename*⟧

In this example, *drive* is a drive letter, *directory* is a valid directory name, and *filename* is a valid filename that must contain at least one wildcard character. The wildcard characters are a question mark (?), meaning "match any character," and an asterisk (*), meaning "match any number of characters."

If the *lpPathSpec* parameter includes a drive and/or directory name, the current drive and directory are changed to the designated drive and directory before the list box is filled. The text control identified by the *nIDStaticPath* parameter is also updated with the new drive and/or directory name.

After the list box is filled, *lpPathSpec* is updated by removing the drive and/or directory portion of the pathname.

DlgDirList sends LB_RESETCONTENT and LB_DIR messages to the list box.

Parameter	Type/Description
hDlg	**HWND** Identifies the dialog box that contains the list box.
lpPathSpec	**LPSTR** Points to a pathname string. The string must be a null-terminated character string.
nIDListBox	**int** Specifies the identifier of a list-box control. If *nIDListBox* is zero, **DlgDirList** assumes that no list box exists and does not attempt to fill it.
nIDStaticPath	**int** Specifies the identifier of the static-text control used for displaying the current drive and directory. If *nIDStaticPath* is zero, **DlgDirList** assumes that no such text control is present.
wFiletype	**WORD** Specifies DOS file attributes of the files to be displayed. It can be any combination of the values given in Table 4.6, "DOS File Attributes." Values can be combined by using the bitwise OR operator.

D

Return Value

The return value specifies the outcome of the function. It is nonzero if a listing was made, even an empty listing. A zero return value implies that the input string did not contain a valid search path.

The *wFiletype* parameter specifies the DOS attributes of the files to be listed. Table 4.6 describes these attributes.

Table 4.6 DOS File Attributes

Attribute Value	Meaning
0x0000	Read/write data files with no additional attributes
0x0001	Read-only files
0x0002	Hidden files
0x0004	System files
0x0010	Subdirectories
0x0020	Archives
0x2000	LB_DIR flag[1]
0x4000	Drives
0x8000	Exclusive bit[2]

[1] If the LB_DIR flag is set, Windows places the messages generated by **DlgDirList** in the application's queue; otherwise they are sent directly to the dialog function.

[2] If the exclusive bit is set, only files of the specified type are listed. Otherwise, files of the specified type are listed in addition to normal files.

DlgDirListComboBox 3.0

Syntax **int DlgDirListComboBox**(*hDlg*, *lpPathSpec*, *nIDComboBox*, *nIDStaticPath*, *wFiletype*)

This function fills the list box of a combo-box control with a file or directory listing. It fills the list box of the combo box specified by the *nIDComboBox* parameter with the names of all files matching the pathname given by the *lpPathSpec* parameter.

The **DlgDirListComboBox** function shows subdirectories enclosed in square brackets ([ ]), and shows drives in the form [–*x*–], where *x* is the drive letter.

The *lpPathSpec* parameter has the following form:

⟦*drive*:⟧ ⟦ ⟦\⟧*directory*⟦*directory*⟧...\⟧ ⟦*filename*⟧

In this example, *drive* is a drive letter, *directory* is a valid directory name, and *filename* is a valid filename that must contain at least one wildcard character. The wildcard characters are a question mark (?), meaning "match any character," and an asterisk (*), meaning "match any number of characters."

If the *lpPathSpec* parameter includes a drive and/or directory name, the current drive and directory are changed to the designated drive and directory before the list box is filled. The text control identified by the *nIDStaticPath* parameter is also updated with the new drive and/or directory name.

After the combo-box list box is filled, *lpPathSpec* is updated by removing the drive and/or directory portion of the pathname.

DlgDirListComboBox sends CB_RESETCONTENT and CB_DIR messages to the combo box.

Parameter	Type/Description
hDlg	**HWND** Identifies the dialog box that contains the combo box.
lpPathSpec	**LPSTR** Points to a pathname string. The string must be a null-terminated character string.
nIDComboBox	**int** Specifies the identifier of a combo-box control in a dialog box. If *nIDComboBox* is zero, **DlgDirListComboBox** assumes that no combo box exists and does not attempt to fill it.
nIDStaticPath	**int** Specifies the identifier of the static-text control used for displaying the current drive and directory. If *nIDStaticPath* is zero, **DlgDirListComboBox** assumes that no such text control is present.

Parameter	Type/Description
wFiletype	**WORD** Specifies DOS file attributes of the files to be displayed. It can be any combination of the values given in Table 4.6, "DOS File Attributes." Refer to the description of the **DlgDirList** function for this table. Values can be combined by using the bitwise OR operator.

Return Value

The return value specifies the outcome of the function. It is nonzero if a listing was made, even an empty listing. A zero return value implies that the input string did not contain a valid search path.

D

DlgDirSelect

Syntax

BOOL DlgDirSelect(*hDlg*, *lpString*, *nIDListBox*)

This function retrieves the current selection from a list box. It assumes that the list box has been filled by the **DlgDirList** function and that the selection is a drive letter, a file, or a directory name.

The **DlgDirSelect** function copies the selection to the buffer given by the *lpString* parameter. If the current selection is a directory name or drive letter, **DlgDirSelect** removes the enclosing square brackets (and hyphens, for drive letters) so that the name or letter is ready to be inserted into a new pathname. If there is no selection, *lpString* does not change.

DlgDirSelect sends LB_GETCURSEL and LB_GETTEXT messages to the list box.

Parameter	Type/Description
hDlg	**HWND** Identifies the dialog box that contains the list box.
lpString	**LPSTR** Points to a buffer that is to receive the selected pathname.
nIDListBox	**int** Specifies the integer ID of a list-box control in the dialog box.

Return Value

The return value specifies the status of the current list-box selection. It is nonzero if the current selection is a directory name. Otherwise, it is zero.

Comments

The **DlgDirSelect** function does not allow more than one filename to be returned from a list box.

The list box must not be a multiple-selection list box. If it is, this function will not return a zero value and *lpString* will remain unchanged.

DlgDirSelectComboBox 3.0

Syntax **BOOL DlgDirSelectComboBox**(*hDlg*, *lpString*, *nIDComboBox*)

This function retrieves the current selection from the list box of a combo box. It assumes that the list box has been filled by the **DlgDirListComboBox** function and that the selection is a drive letter, a file, or a directory name.

The **DlgDirSelectComboBox** function copies the selection to the buffer given by the *lpString* parameter. If the current selection is a directory name or drive letter, **DlgDir-SelectComboBox** removes the enclosing square brackets (and hyphens, for drive letters) so that the name or letter is ready to be inserted into a new pathname. If there is no selection, *lpString* does not change.

DlgDirSelectComboBox sends CB_GETCURSEL and CB_GETLBTEXT messages to the combo box.

Parameter	Type/Description
hDlg	**HWND** Identifies the dialog box that contains the combo box.
lpString	**LPSTR** Points to a buffer that is to receive the selected pathname.
nIDComboBox	**int** Specifies the integer ID of the combo-box control in the dialog box.

Return Value The return value specifies the status of the current combo-box selection. It is nonzero if the current selection is a directory name. Otherwise, it is zero.

Comments The **DlgDirSelectComboBox** function does not allow more than one filename to be returned from a combo box.

DOS3Call 3.0

This function allows an application to issue a DOS function-request interrupt 21H. An application can use this function instead of a directly coded DOS 21H interrupt. The **DOS3Call** function executes somewhat faster than the equivalent DOS 21H software interrupt under Windows.

An application can call this function only from an assembly-language routine. It is exported from KERNEL.EXE and is not defined in any Windows include files.

To use this function call, an application should declare it in an assembly-language program as shown:

```
extrn    DOS3Call    :far
```

If the application includes CMACROS.INC, the application declares it as shown:

```
extrnFP  Dos3Call
```

Before calling **DOS3Call**, all registers must be set as for an actual INT 21H. All registers at the function's exit are the same as for the corresponding INT 21H function.

This function has no parameters and returns the registers of the DOS function.

The following is an example of using **DOS3Call**:

```
extrn DOS3Call : far
        .
        .
        .
; set registers
mov     ah, DOSFUNC
cCall DOS3Call
```

DPtoLP

Syntax

BOOL DPtoLP(*hDC*, *lpPoints*, *nCount*)

This function converts device points into logical points. The function maps the coordinates of each point specified by the *lpPoints* parameter from the device coordinate system into GDI's logical coordinate system. The conversion depends on the current mapping mode and the settings of the origins and extents for the device's window and viewport.

Parameter	Type/Description
hDC	**HDC** Identifies the device context.
lpPoints	**LPPOINT** Points to an array of points. Each point must be a **POINT** data structure.
nCount	**int** Specifies the number of points in the array.

Return Value

The return value specifies whether the conversion has taken place. It is nonzero if all points are converted. Otherwise, it is zero.

DrawFocusRect 3.0

Syntax void **DrawFocusRect**(*hDC*, *lpRect*)

This function draws a rectangle in the style used to indicate focus.

Parameter	Type/Description
hDC	**HDC** Identifies the device context.
lpRect	**LPRECT** Points to a **RECT** data structure that specifies the coordinates of the rectangle to be drawn.

Return Value None.

Comments Since this is an XOR function, calling this function a second time with the same rectangle removes the rectangle from the display.

The rectangle drawn by this function cannot be scrolled. To scroll an area containing a rectangle drawn by this function, call **DrawFocusRect** to remove the rectangle from the display, scroll the area, and then call **DrawFocusRect** to draw the rectangle in the new position.

DrawIcon

Syntax BOOL **DrawIcon**(*hDC*, *X*, *Y*, *hIcon*)

This function draws an icon on the specified device. The **DrawIcon** function places the icon's upper-left corner at the location specified by the *X* and *Y* parameters. The location is subject to the current mapping mode of the device context.

Parameter	Type/Description
hDC	**HDC** Identifies the device context for a window.
X	**int** Specifies the logical *x*-coordinate of the upper-left corner of the icon.
Y	**int** Specifies the logical *y*-coordinate of the upper-left corner of the icon.
hIcon	**HICON** Identifies the icon to be drawn.

Return Value The return value specifies the outcome of the function. It is nonzero if the function is successful. Otherwise, it is zero.

Comments The icon resource must have been previously loaded by using the **LoadIcon** function. The MM_TEXT mapping mode must be selected prior to using this function.

DrawMenuBar

Syntax **void DrawMenuBar**(*hWnd*)

This function redraws the menu bar. If a menu bar is changed *after* Windows has created the window, this function should be called to draw the changed menu bar.

Parameter	Type/Description
hWnd	**HWND** Identifies the window whose menu needs redrawing.

Return Value None.

DrawText

Syntax **int DrawText**(*hDC*, *lpString*, *nCount*, *lpRect*, *wFormat*)

This function draws formatted text in the rectangle specified by the *lpRect* parameter. It formats text by expanding tabs into appropriate spaces, justifying text to the left, right, or center of the given rectangle, and breaking text into lines that fit within the given rectangle. The type of formatting is specified by the *wFormat* parameter.

The **DrawText** function uses the device context's selected font, text color, and background color to draw the text. Unless the DT_NOCLIP format is used, **DrawText** clips the text so that the text does not appear outside the given rectangle. All formatting is assumed to have multiple lines unless the DT_SINGLELINE format is given.

Parameter	Type/Description
hDC	**HDC** Identifies the device context.
lpString	**LPSTR** Points to the string to be drawn. If the *nCount* parameter is –1, the string must be null-terminated.
nCount	**int** Specifies the number of bytes in the string. If *nCount* is –1, then *lpString* is assumed to be a long pointer to a null-terminated string and **DrawText** computes the character count automatically.

Parameter	Type/Description
lpRect	**LPRECT** Points to a **RECT** data structure that contains the rectangle (in logical coordinates) in which the text is to be formatted.
wFormat	**WORD** Specifies the method of formatting the text. It can be a combination of the values given in Table 4.7, "DrawText Formats."

Return Value The return value specifies the height of the text.

Comments If the selected font is too large for the specified rectangle, the **DrawText** function does not attempt to substitute a smaller font.

Table 4.7 lists the values for the *wFormat* parameter. These values can be combined by using the bitwise OR operator. Note that the DT_CALCRECT, DT_EXTERNALLEADING, DT_INTERNAL, DT_NOCLIP, and DT_NOPREFIX values cannot be used with the DT_TABSTOP value.

Table 4.7 DrawText Formats

Value	Meaning
DT_BOTTOM	Specifies bottom-justified text. This value must be combined with DT_SINGLELINE.
DT_CALCRECT	Determines the width and height of the rectangle. If there are multiple lines of text, **DrawText** will use the width of the rectangle pointed to by the *lpRect* parameter and extend the base of the rectangle to bound the last line of text. If there is only one line of text, **DrawText** will modify the right side of the rectangle so that it bounds the last character in the line. In either case, **DrawText** returns the height of the formatted text but does not draw the text.
DT_CENTER	Centers text horizontally.
DT_EXPANDTABS	Expands tab characters. The default number of characters per tab is eight.
DT_EXTERNALLEADING	Includes the font external leading in line height. Normally, external leading is not included in the height of a line of text.
DT_LEFT	Aligns text flush-left.
DT_NOCLIP	Draws without clipping. **DrawText** is somewhat faster when DT_NOCLIP is used.

Table 4.7 DrawText Formats (*continued*)

Value	Meaning
DT_NOPREFIX	Turns off processing of prefix characters. Normally, **DrawText** interprets the mnemonic-prefix character "&" as a directive to underscore the character that follows, and the mnemonic-prefix characters "&&" as a directive to print a single "&". By specifying DT_NOPREFIX, this processing is turned off.
DT_RIGHT	Aligns text flush-right.
DT_SINGLELINE	Specifies single line only. Carriage returns and linefeeds do not break the line.
DT_TABSTOP	Sets tab stops. The high-order byte of the *wFormat* parameter is the number of characters for each tab. The default number of characters per tab is eight.
DT_TOP	Specifies top-justified text (single line only).
DT_VCENTER	Specifies vertically centered text (single line only).
DT_WORDBREAK	Specifies word breaking. Lines are automatically broken between words if a word would extend past the edge of the rectangle specified by the *lpRect* parameter. A carriage return/line sequence will also break the line.

Ellipse

Syntax

BOOL Ellipse(*hDC*, *X1*, *Y1*, *X2*, *Y2*)

This function draws an ellipse. The center of the ellipse is the center of the bounding rectangle specified by the *X1*, *Y1*, *X2*, and *Y2* parameters. The ellipse border is drawn with the current pen, and the interior is filled with the current brush.

If the bounding rectangle is empty, nothing is drawn.

Parameter	Type/Description
hDC	**HDC** Identifies the device context.
X1	**int** Specifies the logical *x*-coordinate of the upper-left corner of the bounding rectangle.
Y1	**int** Specifies the logical *y*-coordinate of the upper-left corner of the bounding rectangle.
X2	**int** Specifies the logical *x*-coordinate of the lower-right corner of the bounding rectangle.
Y2	**int** Specifies the logical *y*-coordinate of the lower-right corner of the bounding rectangle.

Return Value

The return value specifies whether the ellipse is drawn. It is nonzero if the ellipse is drawn. Otherwise, it is zero.

Comments

The width of the rectangle, specified by the absolute value of $X2 - X1$, must not exceed 32,767 units. This limit applies to the height of the rectangle as well.

The current position is neither used nor updated by this function.

EmptyClipboard

Syntax

BOOL EmptyClipboard( )

This function empties the clipboard and frees handles to data in the clipboard. It then assigns ownership of the clipboard to the window that currently has the clipboard open.

This function has no parameters.

Return Value The return value specifies the status of the clipboard. It is nonzero if the clipboard is emptied. It is zero if an error occurs.

Comments The clipboard must be open when the **EmptyClipboard** function is called.

EnableHardwareInput

Syntax BOOL **EnableHardwareInput**(*bEnableInput*)

This function disables mouse and keyboard input. The input is saved if the *bEnableInput* parameter is TRUE and discarded if it is FALSE.

Parameter	Type/Description
bEnableInput	**BOOL** Specifies that the function should save input if the *bEnableInput* parameter is set to a nonzero value; specifies that the function should discard input if the *bEnableInput* parameter is set to zero.

Return Value The return value specifies whether mouse and keyboard input is disabled. It is nonzero if input was previously enabled. Otherwise, it is zero. The default return value is nonzero (TRUE).

EnableMenuItem

Syntax BOOL **EnableMenuItem**(*hMenu*, *wIDEnableItem*, *wEnable*)

This function enables, disables, or grays a menu item.

Parameter	Type/Description
hMenu	**HMENU** Specifies the menu.
wIDEnableItem	**WORD** Specifies the menu item to be checked. The *wIDEnableItem* parameter can specify pop-up menu items as well as menu items.
wEnable	**WORD** Specifies the action to take. It can be a combination of MF_DISABLED, MF_ENABLED, or MF_GRAYED, with MF_BYCOMMAND or MF_BYPOSITION. These values can be combined by using the bitwise OR operator. These values have the following meanings:

Parameter	Type/Description	
	Value	**Meaning**
	MF_BYCOMMAND	Specifies that the *wIDEnableItem* parameter gives the menu item ID (MF_BYCOMMAND is the default ID).
	MF_BYPOSITION	Specifies that the *wIDEnableItem* parameter gives the position of the menu item (the first item is at position zero).
	MF_DISABLED	Menu item is disabled.
	MF_ENABLED	Menu item is enabled.
	MF_GRAYED	Menu item is grayed.

Return Value The return value specifies the previous state of the menu item. The return value is –1 if the menu item does not exist.

Comments To disable or enable input to a menu bar, see the WM_SYSCOMMAND message.

EnableWindow

Syntax BOOL **EnableWindow**(*hWnd*, *bEnable*)

This function enables or disables mouse and keyboard input to the specified window or control. When input is disabled, input such as mouse clicks and key presses are ignored by the window. When input is enabled, all input is processed.

The **EnableWindow** function enables mouse and keyboard input to a window if the *bEnable* parameter is nonzero, and disables it if *bEnable* is zero.

Parameter	Type/Description
hWnd	**HWND** Identifies the window to be enabled or disabled.
bEnable	**BOOL** Specifies whether the given window is to be enabled or disabled.

Return Value The return value specifies the outcome of the function. It is nonzero if the window is enabled or disabled as specified. It is zero if an error occurs.

Comments A window must be enabled before it can be activated. For example, if an application is displaying a modeless dialog box and has disabled its main window, the main window must be enabled before the dialog box is destroyed. Otherwise, another window will get the input focus and be activated. If a child window is disabled, it is ignored when Windows tries to determine which window should get mouse messages.

Initially, all windows are enabled by default. **EnableWindow** must be used to disable a window explicitly.

EndDeferWindowPos 3.0

Syntax **void EndDeferWindowPos**(*hWinPosInfo*)

This function simultaneously updates the position and size of one or more windows in a single screen-refresh cycle. The *hWinPosInfo* parameter identifies a multiple window-position data structure that contains the update information for the windows. The **DeferWindowPos** function stores the update information in the data structure; the **BeginDeferWindowPos** function creates the initial data structure used by these functions.

Parameter	Type/Description
hWinPosInfo	**HANDLE** Identifies a multiple window-position data structure that contains size and position information for one or more windows. This structure is returned by the **BeginDeferWindowPos** function or the most recent call to the **DeferWindowPos** function.

Return Value None.

EndDialog

Syntax **void EndDialog**(*hDlg*, *nResult*)

This function terminates a modal dialog box and returns the given result to the **DialogBox** function that created the dialog box. The **EndDialog** function is required to complete processing whenever the **DialogBox** function is used to create a modal dialog box. The function must be used in the dialog function of the modal dialog box and should not be used for any other purpose.

The dialog function can call **EndDialog** at any time, even during the processing of the WM_INITDIALOG message. If called during the WM_INITDIALOG message, the dialog box is terminated before it is shown or before the input focus is set.

EndDialog does not terminate the dialog box immediately. Instead, it sets a flag that directs the dialog box to terminate as soon as the dialog function ends. The **EndDialog** function returns to the dialog function, so the dialog function must return control to Windows.

Parameter	Type/Description
hDlg	**HWND** Identifies the dialog box to be destroyed.
nResult	**int** Specifies the value to be returned from the dialog box to the **DialogBox** function that created it.

Return Value None.

EndPaint

Syntax **void EndPaint**(*hWnd*, *lpPaint*)

This function marks the end of painting in the given window. The **EndPaint** function is required for each call to the **BeginPaint** function, but only after painting is complete.

Parameter	Type/Description
hWnd	**HWND** Identifies the window that is repainted.
lpPaint	**LPPAINTSTRUCT** Points to a **PAINTSTRUCT** data structure that contains the painting information retrieved by the **BeginPaint** function.

Return Value None.

Comments If the caret was hidden by the **BeginPaint** function, **EndPaint** restores the caret to the screen.

EnumChildWindows

Syntax

BOOL **EnumChildWindows**(*hWndParent*, *lpEnumFunc*, *lParam*)

This function enumerates the child windows that belong to the specified parent window by passing the handle of each child window, in turn, to the application-supplied callback function pointed to by the *lpEnumFunc* parameter.

The **EnumChildWindows** function continues to enumerate windows until the called function returns zero or until the last child window has been enumerated.

Parameter	Type/Description
hWndParent	**HWND** Identifies the parent window whose child windows are to be enumerated.
lpEnumFunc	**FARPROC** Is the procedure-instance address of the callback function.
lParam	**DWORD** Specifies the value to be passed to the callback function for the application's use.

Return Value

The return value specifies nonzero if all child windows have been enumerated. Otherwise, it is zero.

Comments

This function does not enumerate pop-up windows that belong to the *hWndParent* parameter.

The address passed as the *lpEnumFunc* parameter must be created by using the **MakeProcInstance** function.

The callback function must use the Pascal calling convention and must be declared **FAR**.

Callback Function

BOOL FAR PASCAL *EnumFunc*(*hWnd*, *lParam*)
HWND *hWnd*;
DWORD *lParam*;

EnumFunc is a placeholder for the application-supplied function name. The actual name must be exported by including it in an **EXPORTS** statement in the application's module-definition file.

Parameter	Description
hWnd	Identifies the window handle.
lParam	Specifies the long parameter argument of the **EnumChildWindows** function.

Return Value

The callback function should return a nonzero value to continue enumeration; it should return zero to stop enumeration.

EnumClipboardFormats

Syntax WORD **EnumClipboardFormats**(*wFormat*)

This function enumerates the formats found in a list of available formats that belong to the clipboard. On each call to this function, the *wFormat* parameter specifies a known available format, and the function returns the format that appears next in the list. The first format in the list can be retrieved by setting *wFormat* to zero.

Parameter	Type/Description
wFormat	**WORD** Specifies a known format.

Return Value The return value specifies the next known clipboard data format. It is zero if *wFormat* specifies the last format in the list of available formats. It is zero if the clipboard is not open.

Comments Before it enumerates the formats by using the **EnumClipboardFormats** function, an application must open the clipboard by using the **OpenClipboard** function.

The order that an application uses for putting alternative formats for the same data into the clipboard is the same order that the enumerator uses when returning them to the pasting application. The pasting application should use the first format enumerated that it can handle. This gives the donor a chance to recommend formats that involve the least loss of data.

EnumFonts

Syntax

int EnumFonts(*hDC*, *lpFacename*, *lpFontFunc*, *lpData*)

This function enumerates the fonts available on a given device. For each font having the typeface name specified by the *lpFacename* parameter, the **EnumFonts** function retrieves information about that font and passes it to the function pointed to by the *lpFontFunc* parameter. The application-supplied callback function can process the font information as desired. Enumeration continues until there are no more fonts or the callback function returns zero.

Parameter	Type/Description
hDC	**HDC** Identifies the device context.
lpFacename	**LPSTR** Points to a null-terminated character string that specifies the typeface name of the desired fonts. If *lpFacename* is NULL, **EnumFonts** randomly selects and enumerates one font of each available typeface.
lpFontFunc	**FARPROC** Is the procedure-instance address of the callback function. See the following "Comments" section for details.
lpData	**LPSTR** Points to the application-supplied data. The data is passed to the callback function along with the font information.

Return Value

The return value specifies the last value returned by the callback function. Its meaning is user-defined.

Comments

The address passed as the *lpFontFunc* parameter must be created by using the **MakeProcInstance** function.

The callback function must use the Pascal calling convention and must be declared **FAR**.

Callback Function

int FAR PASCAL *FontFunc*(*lpLogFont*, *lpTextMetrics*, *nFontType*, *lpData*)
LPLOGFONT *lpLogFont*;
LPTEXTMETRICS *lpTextMetrics*;
short *nFontType*;
LPSTR *lpData*;

FontFunc is a placeholder for the application-supplied function name. The actual name must be exported by including it in an **EXPORTS** statement in the application's module-definition file.

Parameter	Description
lpLogFont	Points to a **LOGFONT** data structure that contains information about the logical attributes of the font.
lpTextMetrics	Points to a **TEXTMETRIC** data structure that contains information about the physical attributes of the font.
nFontType	Specifies the type of the font.
lpData	Points to the application-supplied data passed by **EnumFonts**.

Return Value

The return value can be any integer.

Comments

The AND (&) operator can be used with the RASTER_FONTTYPE and DEVICE_FONTTYPE constants to determine the font type. The RASTER_FONTTYPE bit of the *FontType* parameter specifies whether the font is a raster or vector font. If the bit is one, the font is a raster font; if zero, it is a vector font. The DEVICE_FONTTYPE bit of *FontType* specifies whether the font is a device- or GDI-based font. If the bit is one, the font is a device-based font; if zero, it is a GDI-based font.

If the device is capable of text transformations (scaling, italicizing, and so on) only the base font will be enumerated. The user must inquire into the device's text-transformation abilities to determine which additional fonts are available directly from the device. GDI can simulate the bold, italic, underlined, and strikeout attributes for any GDI-based font.

EnumFonts only enumerates fonts from the GDI internal table. This does not include fonts that are generated by a device, such as fonts that are transformations of fonts from the internal table. The **GetDeviceCaps** function can be used to determine which transformations a device can perform. This information is available by using the TEXTCAPS index.

GDI can scale GDI-based raster fonts by one to five horizontally and one to eight vertically, unless PROOF_QUALITY is being used.

EnumMetaFile

Syntax BOOL **EnumMetaFile**(*hDC*, *hMF*, *lpCallbackFunc*, *lpClientData*)

This function enumerates the GDI calls within the metafile identified by the *hMF* parameter. The **EnumMetaFile** function retrieves each GDI call within the metafile and passes it to the function pointed to by the *lpCallbackFunc* parameter. This callback function, an application-supplied function, can process each GDI call as desired. Enumeration continues until there are no more GDI calls or the callback function returns zero.

Parameter	Type/Description
hDC	**HDC** Identifies the device context associated with the metafile.
hMF	**LOCALHANDLE** Identifies the metafile.
lpCallbackFunc	**FARPROC** Is the procedure-instance callback function. See the following "Comments" section for details.
lpClientData	**BYTE FAR *** Points to the callback-function data.

Return Value The return value specifies the outcome of the function. It is nonzero if the callback function enumerates all the GDI calls in a metafile; otherwise, it returns zero.

Comments The callback function must use the Pascal calling convention and must be declared **FAR**.

Callback Function

int FAR PASCAL *EnumFunc*(*hDC*, *lpHTable*, *lpMFR*, *nObj*, *lpClientData*)
HDC *hDC*;
LPHANDLETABLE *lpHTable*;
LPMETARECORD *lpMFR*;
int *nObj*;
BYTE FAR * *lpClientData*;

EnumFunc is a placeholder for the application-supplied function name. The actual name must be exported by including it in an **EXPORTS** statement in the application's module-definition file.

Parameter	Description
hDC	Identifies the special device context that contains the metafile.
lpHTable	Points to a table of handles associated with the objects (pens, brushes, and so on) in the metafile.
lpMFR	Points to a metafile record contained in the metafile.
nObj	Specifies the number of objects with associated handles in the handle table.
lpClientData	Points to the application-supplied data.

Return Value

The function can carry out any desired task. It must return a nonzero value to continue enumeration, or a zero value to stop it.

EnumObjects

Syntax

int EnumObjects(*hDC*, *nObjectType*, *lpObjectFunc*, *lpData*)

This function enumerates the pens and brushes available on a device. For each object that belongs to the given style, the callback function is called with the information for that object. The callback function is called until there are no more objects or the callback function returns zero.

Parameter	Type/Description
hDC	**HDC** Identifies the device context.
nObjectType	**int** Specifies the object type. It can be one of the following values: OBJ_BRUSH OBJ_PEN
lpObjectFunc	**FARPROC** Is the procedure-instance address of the application-supplied callback function. See the following "Comments" section for details.
lpData	**LPSTR** Points to the application-supplied data. The data is passed to the callback function along with the object information.

Return Value

The return value specifies the last value returned by the callback function. Its meaning is user-defined.

Comments

The address passed as the *lpObjectFunc* parameter must be created by using the **MakeProcInstance** function.

The callback function must use the Pascal calling convention and must be declared **FAR.**

Callback Function

int FAR PASCAL *ObjectFunc*(*lpLogObject*, *lpData*)
char FAR * *lpLogObject*;
char FAR * *lpData*;

ObjectFunc is a placeholder for the application-supplied function name. The actual name must be exported by including it in an **EXPORTS** statement in the application's module-definition file.

Parameter	Description
lpLogObject	Points to a **LOGPEN** or **LOGBRUSH** data structure that contains information about the logical attributes of the object.
lpData	Points to the application-supplied data passed to the **EnumObjects** function.

EnumProps

Syntax

int **EnumProps**(*hWnd*, *lpEnumFunc*)

This function enumerates all entries in the property list of the specified window. It enumerates the entries by passing them, one by one, to the callback function specified by *lpEnumFunc*. **EnumProps** continues until the last entry is enumerated or the callback function returns zero.

Parameter	Type/Description
hWnd	**HWND** Identifies the window whose property list is to be enumerated.
lpEnumFunc	**FARPROC** Is the procedure-instance address of the callback function. See the following "Comments" section for details.

Return Value

The return value specifies the last value returned by the callback function. It is –1 if the function did not find a property for enumeration.

Comments

An application can remove only those properties which it has added. It should not remove properties added by other applications or by Windows itself.

The following restrictions apply to the callback function:

1. The callback function must not yield control or do anything that might yield control to other tasks.
2. The callback function can call the **RemoveProp** function. However, the **RemoveProp** function can remove only the property passed to the callback function through the callback function's parameters.
3. A callback function should not attempt to add properties.

The address passed in the *lpEnumFunc* parameter must be created by using the **MakeProcInstance** function.

Fixed Data Segments

The callback function must use the Pascal calling convention and must be declared **FAR**. In applications and dynamic libraries with fixed data segments and in dynamic libraries with moveable data segments that do not contain a stack, the callback function must have the form shown below.

Callback Function

int FAR PASCAL *EnumFunc*(*hWnd*, *lpString*, *hData*)
HWND *hWnd*;
LPSTR *lpString*;
HANDLE *hData*;

EnumFunc is a placeholder for the application-supplied function name. The actual name must be exported by including it in an **EXPORTS** statement in the application's module-definition file.

Parameter	Description
hWnd	Identifies a handle to the window that contains the property list.
lpString	Points to the null-terminated character string associated with the data handle when the application called the **SetProp** function to set the property. If the application passed an atom instead of a string to the **SetProp** function, the *lpString* parameter contains the atom in its low-order word, and the high-order word is zero.
hData	Identifies the data handle.

Return Value

The callback function can carry out any desired task. It must return a nonzero value to continue enumeration, or a zero value to stop it.

Moveable Data Segments

The callback function must use the Pascal calling convention and must be declared **FAR**. In applications with moveable data segments and in dynamic libraries whose moveable data segments also contain a stack, the callback function must have the form shown below.

Callback Function

int FAR PASCAL *EnumFunc*(*hWnd*, *nDummy*, *pString*, *hData*)
HWND *hWnd*;
WORD *nDummy*;
PSTR *pString*;
HANDLE *hData*;

EnumFunc is a placeholder for the application-supplied function name. The actual name must be exported by including it in an **EXPORTS** statement in the application's module-definition file.

Parameter	Description
hWnd	Identifies a handle to the window that contains the property list.
nDummy	Specifies a dummy parameter.
pString	Points to the null-terminated character string associated with the data handle when the application called the **SetProp** function to set the property. If the application passed an atom instead of a string to the **SetProp** function, the *pString* parameter contains the atom.
hData	Identifies the data handle.

Return Value

The callback function can carry out any desired task. It should return a nonzero value to continue enumeration, or a zero value to stop it.

Comments

The alternate form above is required since movement of the data will invalidate any long pointer to a variable on the stack, such as the *lpString* parameter. The data segment typically moves if the callback function allocates more space in the local heap than is currently available.

EnumTaskWindows

Syntax **BOOL EnumTaskWindows**(*hTask*, *lpEnumFunc*, *lParam*)

This function enumerates all windows associated with the *hTask* parameter, which is returned by the **GetCurrentTask** function. (A task is any program that executes as an independent unit. All applications are executed as tasks and each instance of an application is a task.) The enumeration terminates when the callback function, pointed to by *lpEnumFunc*, returns FALSE.

Parameter	Type/Description
hTask	**HANDLE** Identifies the specified task. The **GetCurrentTask** function returns this handle.
lpEnumFunc	**FARPROC** Is the procedure-instance address of the window's callback function.
lParam	**DWORD** Specifies the 32-bit value that contains additional parameters that are sent to the callback function pointed to by *lpEnumFunc*.

Return Value

The return value specifies the outcome of the function. It is nonzero if all the windows associated with a particular task are enumerated. Otherwise, it is zero.

Comments

The callback function must use the Pascal calling convention and must be declared **FAR**. The callback function must have the following form:

Callback Function

BOOL FAR PASCAL *EnumFunc*(*hWnd*, *lParam*)
HWND *hWnd*;
DWORD *lParam*;

EnumFunc is a placeholder for the application-supplied function name. The actual name must be exported by including it in an **EXPORTS** statement in the application's module-definition file.

Parameter	Description
hWnd	Identifies a window associated with the current task.
lParam	Specifies the same argument that was passed to the **EnumTaskWindows** function.

Return Value

The callback function can carry out any desired task. It must return a nonzero value to continue enumeration, or a zero value to stop it.

EnumWindows

Syntax

BOOL EnumWindows(*lpEnumFunc*, *lParam*)

This function enumerates all parent windows on the screen by passing the handle of each window, in turn, to the callback function pointed to by the *lpEnumFunc* parameter. Child windows are not enumerated.

The **EnumWindows** function continues to enumerate windows until the called function returns zero or until the last window has been enumerated.

Parameter	Type/Description
lpEnumFunc	**FARPROC** Is the procedure-instance address of the callback function. See the following "Comments" section for details.
lParam	**DWORD** Specifies the value to be passed to the callback function for the application's use.

Return Value

The return value specifies the outcome of the function. It is nonzero if all windows have been enumerated. Otherwise, it is zero.

Comments

The address passed as the *lpEnumFunc* parameter must be created by using the **MakeProcInstance** function.

The callback function must use the Pascal calling convention and must be declared **FAR**. The callback function must have the following form:

Callback Function

BOOL FAR PASCAL *EnumFunc*(*hWnd*, *lParam*)
HWND *hWnd*;
DWORD *lParam*;

EnumFunc is a placeholder for the application-supplied function name. The actual name must be exported by including it in an **EXPORTS** statement in the application's module-definition file.

Parameter	Description
hWnd	Identifies the window handle.
lParam	Specifies the 32-bit argument of the **EnumWindows** function.

Return Value

The function must return a nonzero value to continue enumeration, or zero to stop it.

EqualRect

Syntax **BOOL EqualRect**(*lpRect1*, *lpRect2*)

This function determines whether two rectangles are equal by comparing the coordinates of their upper-left and lower-right corners. If the values of these coordinates are equal, **EqualRect** returns a nonzero value; otherwise, it returns zero.

Parameter	Type/Description
lpRect1	**LPRECT** Points to a **RECT** data structure that contains the upper-left and lower-right corner coordinates of the first rectangle.
lpRect2	**LPRECT** Points to a **RECT** data structure that contains the upper-left and lower-right corner coordinates of the second rectangle.

Return Value The return value specifies whether the specified rectangles are equal. It is nonzero if the two rectangles are identical. Otherwise, it is zero.

EqualRgn

Syntax **BOOL EqualRgn**(*hSrcRgn1*, *hSrcRgn2*)

This function checks the two given regions to determine whether they are identical.

Parameter	Type/Description
hSrcRgn1	**HRGN** Identifies a region.
hSrcRgn2	**HRGN** Identifies a region.

Return Value The return value specifies whether the specified regions are equal. It is nonzero if the two regions are equal. Otherwise, it is zero.

Escape

Syntax **int Escape**(*hDC*, *nEscape*, *nCount*, *lpInData*, *lpOutData*)

This function allows applications to access facilities of a particular device that are not directly available through GDI. Escape calls made by an application are translated and sent to the device driver.

Parameter	Type/Description
hDC	**HDC** Identifies the device context.
nEscape	**int** Specifies the escape function to be performed. For a complete list of escape functions, see Chapter 12, "Printer Escapes," in *Reference, Volume 2*.
nCount	**int** Specifies the number of bytes of data pointed to by the *lpIn-Data* parameter.
lpInData	**LPSTR** Points to the input data structure required for this escape.
lpOutData	**LPSTR** Points to the data structure to receive output from this escape. The *lpOutData* parameter should be NULL if no data are returned.

Return Value The return value specifies the outcome of the function. It is positive if the function is successful except for the QUERYESCSUPPORT escape, which only checks for implementation. The return value is zero if the escape is not implemented. A negative value indicates an error. The following list shows common error values:

Value	Meaning
SP_ERROR	General error.
SP_OUTOFDISK	Not enough disk space is currently available for spooling, and no more space will become available.
SP_OUTOFMEMORY	Not enough memory is available for spooling.
SP_USERABORT	User terminated the job through the Print Manager.

EscapeCommFunction

Syntax **int EscapeCommFunction**(*nCid, nFunc*)

This function directs the communication device specified by the *nCid* parameter to carry out the extended function specified by the *nFunc* parameter.

Parameter	Type/Description
nCid	**int** Specifies the communication device to carry out the extended function. The **OpenComm** function returns this value.

Parameter	Type/Description
nFunc	**int** Specifies the function code of the extended function. It can be any one of the following values:

Value	Description
CLRDTR	Clears the data-terminal-ready (DTR) signal.
CLRRTS	Clears the request-to-send (RTS) signal.
RESETDEV	Resets the device if possible.
SETDTR	Sends the data-terminal-ready (DTR) signal.
SETRTS	Sends the request-to-send (RTS) signal.
SETXOFF	Causes transmission to act as if an XOFF character has been received.
SETXON	Causes transmission to act as if an XON character has been received.

Return Value The return value specifies the result of the function. It is zero if it is successful. It is negative if the *nFunc* parameter does not specify a valid function code.

ExcludeClipRect

Syntax **int ExcludeClipRect**(*hDC*, *X1*, *Y1*, *X2*, *Y2*)

This function creates a new clipping region that consists of the existing clipping region minus the specified rectangle.

Parameter	Type/Description
hDC	**HDC** Identifies the device context.
X1	**int** Specifies the logical *x*-coordinate of the upper-left corner of the rectangle.
Y1	**int** Specifies the logical *y*-coordinate of the upper-left corner of the rectangle.
X2	**int** Specifies the logical *x*-coordinate of the lower-right corner of the rectangle.

Parameter	Type/Description
Y2	**int** Specifies the logical *y*-coordinate of the lower-right corner of the rectangle.

Return Value The return value specifies the new clipping region's type. It can be any one of the following values:

Value	Meaning
COMPLEXREGION	The region has overlapping borders.
ERROR	No region was created.
NULLREGION	The region is empty.
SIMPLEREGION	The region has no overlapping borders.

Comments The width of the rectangle, specified by the absolute value of *X2* – *X1*, must not exceed 32,767 units. This limit applies to the height of the rectangle as well.

ExcludeUpdateRgn

Syntax **int ExcludeUpdateRgn**(*hDC*, *hWnd*)

This function prevents drawing within invalid areas of a window by excluding an updated region in the window from a clipping region.

Parameter	Type/Description
hDC	**HANDLE** Identifies the device context associated with the clipping region.
hWnd	**HWND** Identifies the window being updated.

Return Value This value specifies the type of resultant region. It can be any one of the following values:

Value	Meaning
COMPLEXREGION	The region has overlapping borders.
ERROR	No region was created.

Value	Meaning
NULLREGION	The region is empty.
SIMPLEREGION	The region has no overlapping borders.

ExitWindows 3.0

Syntax **BOOL ExitWindows**(*dwReserved*, *wReturnCode*)

This function initiates the standard Windows shutdown procedure. If all applications agree to terminate, the Windows session is terminated and control returns to DOS. Windows sends the WM_QUERYENDSESSION message to notify all applications that a request has been made to terminate Windows. If all applications agree to terminate, Windows sends the WM_ENDSESSION message to all applications before exiting.

Parameter	Type/Description
dwReserved	**DWORD** Is reserved and should be set to zero.
wReturnCode	**WORD** Specifies the return value to be passed to DOS when Windows exits.

Return Value The return value is FALSE if one or more applications refused to terminate. The function does not return if all applications agree to be terminated.

ExtDeviceMode 3.0

Syntax **int ExtDeviceMode**(*hWnd*, *hDriver*, *lpDevModeOutput*, *lpDeviceName*, *lpPort*, *lpDevModeInput*, *lpProfile*, *wMode*)

This function retrieves or modifies device initialization information for a given printer driver, or displays a driver-supplied dialog box for configuring the printer driver. Printer drivers that support device initialization by applications export this **ExtDeviceMode** so that applications can call it.

Parameter	Type/Description
hWnd	**HWND** Identifies a window. If the application calls **ExtDeviceMode** to display a dialog box, the specified window is the parent of the dialog box.

Parameter	Type/Description
hDriver	**HANDLE** Identifies the device-driver module. The **GetModuleHandle** function or **LoadLibrary** function returns a module handle.
lpDevModeOutput	**DEVMODE FAR *** Points to a **DEVMODE** data structure. The driver writes the initialization information supplied in the *lpDevModeInput* parameter to this structure.
lpDeviceName	**LPSTR** Points to a null-terminated character string that contains the name of the printer device, such as "PCL/HP LaserJet."
lpPort	**LPSTR** Points to a null-terminated character string that contains the name of the port to which the device is connected, such as LPT1:.
lpDevModeInput	**DEVMODE FAR *** Points to a **DEVMODE** data structure that supplies initialization information to the printer driver.
lpProfile	**LPSTR** Points to a null-terminated string that contains the name of the initialization file which initialization information is recorded in and read from. If this parameter is NULL, WIN.INI is the default.
wMode	**WORD** Specifies a mask of values which determine the types of operations the function will perform. If *wMode* is zero, **ExtDeviceMode** returns the number of bytes required by the printer device driver's **DEVMODE** structure. Otherwise, *wMode* must be one or more of the following values:

Value	Meaning
DM_COPY	Writes the printer driver's current print settings to the **DEVMODE** data structure identified by **lpDevModeOutput.** The calling application must allocate a buffer sufficiently large to contain the information. If this bit is clear, *lpDevModeOutput* can be NULL.

Parameter	Type/Description	
	Value	**Meaning**
	DM_MODIFY	Changes the printer driver's current print settings to match the partial initialization data in the **DEVMODE** data structure identified by *lpDevModeInput* before prompting, copying, or updating.
	DM_PROMPT	Presents the printer driver's Print Setup dialog box and then changes the current print settings to those the user specifies.
	DM_UPDATE	Writes the printer driver's current print settings to the printer environment and the WIN.INI initialization file.

Return Value

If the *wMode* parameter is zero, the return value is the size of the **DEVMODE** data structure required to contain the printer driver initialization data. If the function displays the initialization dialog box, the return value is either IDOK or IDCANCEL, depending on which button the user selected. If the function does not display the dialog box and was successful, the return value is IDOK. The return value is less than zero if the function failed.

Comments

The **ExtDeviceMode** function is actually part of the printer's device driver, and not part of GDI. To call this function, the application must include the DRIVINT.H file, load the printer device driver, and retrieve the address of the function by using the **GetProcAddress** function. The application can then use the address to set up the printer.

An application can set the *wMode* parameter to DM_COPY to obtain a **DEVMODE** data structure filled in with the printer driver's initialization data. The application can then pass this data structure to the **CreateDC** function to set a private environment for the printer device context.

ExtFloodFill [3.0]

Syntax **BOOL** **ExtFloodFill**(*hDC*, *X*, *Y*, *crColor*, *wFillType*)

This function fills an area of the display surface with the current brush.

If *wFillType* is set to FLOODFILLBORDER, the area is assumed to be completely bounded by the color specified by the *crColor* parameter. The **ExtFloodFill** function begins at the point specified by the *X* and *Y* parameters and fills in all directions to the color boundary.

If *wFillType* is set to FLOODFILLSURFACE, the **ExtFloodFill** function begins at the point specified by *X* and *Y* and continues in all directions, filling all adjacent areas containing the color specified by *crColor*.

Parameter	Type/Description
hDC	**HDC** Identifies the device context.
X	**int** Specifies the logical *x*-coordinate of the point where filling begins.
Y	**int** Specifies the logical *y*-coordinate of the point where filling begins.
crColor	**COLORREF** Specifies the color of the boundary or of the area to be filled. The interpretation of *crColor* depends on the value of the *wFillType* parameter.
wFillType	**WORD** Specifies the type of flood fill to be performed. It must be one of the following values:

Value	Meaning
FLOODFILLBORDER	The fill area is bounded by the color specified by *crColor*. This style is identical to the filling performed by the **FloodFill** function.
FLOODFILLSURFACE	The fill area is defined by the color specified by *crColor*. Filling continues outward in all directions as long as the color is encountered. This style is useful for filling areas with multi-colored boundaries.

Return Value

The return value specifies the outcome of the function. It is nonzero if the function is successful. It is zero if:

- The filling could not be completed
- The given point has the boundary color specified by *crColor* (if FLOOD-FILLBORDER was requested)
- The given point does not have the color specified by *crColor* (if FLOOD-FILLSURFACE was requested)
- The point is outside the clipping region

Comments

Only memory device contexts and devices that support raster-display technology support the **ExtFloodFill** function. For more information, see the RC_BITBLT raster capability in the **GetDeviceCaps** function, later in this chapter.

ExtTextOut

Syntax

BOOL ExtTextOut(*hDC*, *X*, *Y*, *wOptions*, *lpRect*, *lpString*, *nCount*, *lpDx*)

This function writes a character string, within a rectangular region on the specified display, using the currently selected font. The rectangular region can be opaque (filled with the current background color) and it can be a clipping region.

Parameter	Type/Description
hDC	**HDC** Identifies the device context.
X	**int** Specifies the logical *x*-coordinate of the origin of the character cell for the first character in the specified string.
Y	**int** Specifies the logical *y*-coordinate of the origin of the character cell for the first character in the specified string.
wOptions	**WORD** Specifies the rectangle type. It can be one or both of the following values, or neither: ETO_CLIPPED ETO_OPAQUE The ETO_CLIPPED value specifies that Windows will clip text to the rectangle. The ETO_OPAQUE value specifies that the current background color fills the rectangle.
lpRect	**LPRECT** Points to a **RECT** data structure. The *lpRect* parameter can be NULL.

Parameter	Type/Description
lpString	**LPSTR** Points to the specified character string.
nCount	**int** Specifies the number of characters in the string.
lpDx	**LPINT** Points to an array of values that indicate the distance between origins of adjacent character cells. For instance, *lpDx*[*i*] logical units will separate the origins of character cell *i* and character cell *i* + 1.

Return Value The return value specifies whether or not the string is drawn. It is nonzero if the string is drawn. Otherwise, it is zero.

Comments If *lpDx* is NULL, the function uses the default spacing between characters.

The character-cell origins and the contents of the array pointed to by the *lpDx* parameter are given in logical units. A character-cell origin is defined as the upper-left corner of the character cell.

By default, the current position is not used or updated by this function. However, an application can call the **SetTextAlign** function with the *wFlags* parameter set to TA_UPDATECP to permit Windows to use and update the current position each time the application calls **ExtTextOut** for a given device context. When this flag is set, Windows ignores the *X* and *Y* parameters on subsequent **ExtTextOut** calls.

FatalAppExit 3.0

Syntax VOID **FatalAppExit**(*wAction*, *lpMessageText*)

This function displays a message containing the text specified by the *lpMessageText* parameter and terminates the application when the message box is closed. When called under the debugging version of Windows, the message box gives the user the opportunity to terminate the application or to return to the caller.

Parameter	Type/Description
wAction	**WORD** Is reserved and must be set to 0.
lpMessageText	**LPSTR** Points to a character string that is displayed in the message box. The message is displayed on a single line. To accommodate low-resolution displays, the string should be no more than 35 characters in length.

Return Value None.

Comments An application that encounters an unexpected error should terminate by freeing all its memory and then returning from its main message loop. It should call **FatalAppExit** only when it is not capable of terminating any other way. **FatalAppExit** may not always free an application's memory or close its files, and it may cause a general failure of Windows.

FatalExit

Syntax void **FatalExit**(*Code*)

This function displays the current state of Windows on the debugging monitor and prompts for instructions on how to proceed. The display includes an error code, the *Code* parameter, followed by a symbolic stack trace, showing the flow of execution up to the point of call.

An application should call this function only for debugging purposes; it should not call the function in a retail version of the application. Calling this function in the retail version will terminate the application.

Parameter	Type/Description
Code	**int** Specifies the error code to be displayed.

Return Value None.

Comments

The **FatalExit** function prompts the user to respond to an "Abort, Break or Ignore" message. **FatalExit** processes the response as follows:

Response	Description
A (Abort)	Terminates Windows.
B (Break)	Simulates a non-maskable interrupt (NMI) to enter the debugger.
I (Ignore)	Returns to the caller.

The **FatalExit** function is for debugging only.

An application should call this function whenever the application detects a fatal error. All input and output is received and transmitted through the computer's auxiliary port (AUX) or through the debugger if a debugger is installed.

FillRect

Syntax

int FillRect(*hDC*, *lpRect*, *hBrush*)

This function fills a given rectangle by using the specified brush. The **FillRect** function fills the complete rectangle, including the left and top borders, but does not fill the right and bottom borders.

Parameter	Type/Description
hDC	**HDC** Identifies the device context.
lpRect	**LPRECT** Points to a **RECT** data structure that contains the logical coordinates of the rectangle to be filled.
hBrush	**HBRUSH** Identifies the brush used to fill the rectangle.

Return Value

Although the **FillRect** function return type is an integer, the return value is not used and has no meaning.

Comments

The brush must have been created previously by using either the **CreateHatchBrush**, **CreatePatternBrush**, or **CreateSolidBrush** function, or retrieved using the **GetStockObject** function.

When filling the specified rectangle, the **FillRect** function does not include the rectangle's right and bottom sides. GDI fills a rectangle up to, but does not include, the right column and bottom row, regardless of the current mapping mode.

FillRect compares the values of the **top**, **bottom**, **left**, and **right** fields of the specified rectangle. If **bottom** is less than or equal to **top**, or if **right** is less than or equal to **left**, the rectangle is not drawn.

FillRgn

Syntax

BOOL FillRgn(*hDC*, *hRgn*, *hBrush*)

This function fills the region specified by the *hRgn* parameter with the brush specified by the *hBrush* parameter.

Parameter	Type/Description
hDC	**HDC** Identifies the device context.
hRgn	**HRGN** Identifies the region to be filled. The coordinates for the given region are specified in device units.
hBrush	**HBRUSH** Identifies the brush to be used to fill the region.

Return Value

The return value specifies the outcome of the function. It is nonzero if the function is successful. Otherwise, it is zero.

FindAtom

Syntax

ATOM FindAtom(*lpString*)

This function searches the atom table for the character string pointed to by the *lpString* parameter and retrieves the atom associated with that string.

Parameter	Type/Description
lpString	**LPSTR** Points to the character string to be searched for. The string must be null-terminated.

Return Value

The return value identifies the atom associated with the given string. It is NULL if the string is not in the table.

FindResource

Syntax HANDLE **FindResource**(*hInstance, lpName, lpType*)

This function determines the location of a resource in the specified resource file. The *lpName* and *lpType* parameters define the resource name and type, respectively.

Parameter	Type/Description
hInstance	**HANDLE** Identifies the instance of the module whose executable file contains the resource.
lpName	**LPSTR** Points to a null-terminated character string that represents the name of the resource.
lpType	**LPSTR** Points to a null-terminated character string that represents the type name of the resource. For predefined resource types, the *lpType* parameter should be one of the following values:

Value	Meaning
RT_ACCELERATOR	Accelerator table
RT_BITMAP	Bitmap resource
RT_DIALOG	Dialog box
RT_FONT	Font resource
RT_FONTDIR	Font directory resource
RT_MENU	Menu resource
RT_RCDATA	User-defined resource (raw data)

Return Value The return value identifies the named resource. It is NULL if the requested resource cannot be found.

Comments An application must not call **FindResource** and the **LoadResource** function to load cursor, icon, and string resources. Instead, it must load these resources by calling the following functions:

- **LoadCursor**
- **LoadIcon**
- **LoadString**

An application can call **FindResource** and **LoadResource** to load other predefined resource types. However, it is recommended that the application load the corresponding resources by calling the following functions:

- **LoadAccelerators**
- **LoadBitmap**
- **LoadMenu**

If the high-order word of the *lpName* or *lpType* parameter is zero, the low-order word specifies the integer ID of the name or type of the given resource. Otherwise, the parameters are long pointers to null-terminated character strings. If the first character of the string is a pound sign (#), the remaining characters represent a decimal number that specifies the integer ID of the resource's name or type. For example, the string #258 represents the integer ID 258.

To reduce the amount of memory required for the resources used by an application, the application should refer to the resources by integer ID instead of by name.

FindWindow

Syntax

HWND FindWindow(*lpClassName, lpWindowName*)

This function returns the handle of the window whose class is given by the *lpClassName* parameter and whose window name, or caption, is given by the *lpWindowName* parameter.

Parameter	Type/Description
lpClassName	**LPSTR** Points to a null-terminated character string that specifies the window's class name. If *lpClassName* is NULL, all class names match.
lpWindowName	**LPSTR** Points to a null-terminated character string that specifies the window name (the window's text caption). If *lpWindowName* is NULL, all window names match.

Return Value

The return value identifies the window that has the specified class name and window name. It is NULL if no such window is found.

FlashWindow

Syntax BOOL **FlashWindow**(*hWnd*, *bInvert*)

This function "flashes" the given window once. Flashing a window means changing the appearance of its caption bar as if the window were changing from inactive to active status, or vice versa. (An inactive caption bar changes to an active caption bar; an active caption bar changes to an inactive caption bar.)

Typically, a window is flashed to inform the user that the window requires attention, but that it does not currently have the input focus.

Parameter	Type/Description
hWnd	**HWND** Identifies the window to be flashed. The window can be either open or iconic.
bInvert	**BOOL** Specifies whether the window is to be flashed or returned to its original state. The window is flashed from one state to the other if the *bInvert* parameter is nonzero. If the *bInvert* parameter is zero, the window is returned to its original state (either active or inactive).

Return Value The return value specifies the window's state before call to the **FlashWindow** function. It is nonzero if the window was active before the call. Otherwise, it is zero.

Comments The **FlashWindow** function flashes the window only once; for successive flashing, the application should create a system timer.

The *bInvert* parameter should be zero only when the window is getting the input focus and will no longer be flashing; it should be nonzero on successive calls while waiting to get the input focus.

This function always returns a nonzero value for iconic windows. If the window is iconic, **FlashWindow** will simply flash the icon; *bInvert* is ignored for iconic windows.

FloodFill

Syntax BOOL **FloodFill**(*hDC*, *X*, *Y*, *crColor*)

This function fills an area of the display surface with the current brush. The area is assumed to be bounded as specified by the *crColor* parameter. The **FloodFill** function begins at the point specified by the *X* and *Y* parameters and continues in all directions to the color boundary.

Parameter	Type/Description
hDC	**HDC** Identifies the device context.
X	**int** Specifies the logical *x*-coordinate of the point where filling begins.
Y	**int** Specifies the logical *y*-coordinate of the point where filling begins.
crColor	**COLORREF** Specifies the color of the boundary.

Return Value The return value specifies the outcome of the function. It is nonzero if the function is successful. It is zero if the filling could not be completed, the given point has the boundary color specified by *crColor*, or the point is outside the clipping region.

Comments Only memory device contexts and devices that support raster-display technology support the **FloodFill** function. For more information, see the RC_BITBLT raster capability in the **GetDeviceCaps** function, later in this chapter.

FlushComm

Syntax **int FlushComm**(*nCid*, *nQueue*)

This function flushes all characters from the transmit or receive queue of the communication device specified by the *nCid* parameter. The *nQueue* parameter specifies which queue is to be flushed.

Parameter	Type/Description
nCid	**int** Specifies the communication device to be flushed. The **OpenComm** function returns this value.
nQueue	**int** Specifies the queue to be flushed. If *nQueue* is zero, the transmit queue is flushed. If it is 1, the receive queue is flushed.

Return Value The return value specifies the result of the function. It is zero if it is successful. It is negative if *nCid* is not a valid device, or if *nQueue* is not a valid queue.

FrameRect

Syntax

int FrameRect(*hDC*, *lpRect*, *hBrush*)

This function draws a border around the rectangle specified by the *lpRect* parameter. The **FrameRect** function uses the given brush to draw the border. The width and height of the border is always one logical unit.

Parameter	Type/Description
hDC	**HDC** Identifies the device context of the window.
lpRect	**LPRECT** Points to a **RECT** data structure that contains the logical coordinates of the upper-left and lower-right corners of the rectangle.
hBrush	**HBRUSH** Identifies the brush to be used for framing the rectangle.

Return Value

Although the return value type is integer, its contents should be ignored.

Comments

The brush identified by the *hBrush* parameter must have been created previously by using the **CreateHatchBrush**, **CreatePatternBrush**, or **CreateSolidBrush** function.

If the **bottom** field is less than or equal to the **top** field, or if **right** is less than or equal to **left**, the rectangle is not drawn.

FrameRgn

Syntax

BOOL FrameRgn(*hDC*, *hRgn*, *hBrush*, *nWidth*, *nHeight*)

This function draws a border around the region specified by the *hRgn* parameter, using the brush specified by the *hBrush* parameter. The *nWidth* parameter specifies the width of the border in vertical brush strokes; the *nHeight* parameter specifies the height in horizontal brush strokes.

Parameter	Type/Description
hDC	**HDC** Identifies the device context.
hRgn	**HANDLE** Identifies the region to be enclosed in a border. The coordinates for the given region are specified in device units.

Parameter	Type/Description
hBrush	**HBRUSH** Identifies the brush to be used to draw the border.
nWidth	**int** Specifies the width in vertical brush strokes (in logical units).
nHeight	**int** Specifies the height in horizontal brush strokes (in logical units).

Return Value The return value specifies the outcome of the function. It is nonzero if the function is successful. Otherwise, it is zero.

FreeLibrary

Syntax void **FreeLibrary**(*hLibModule*)

This function decreases the reference count of the loaded library module by one. When the reference count reaches zero, the memory occupied by the module is freed.

Parameter	Type/Description
hLibModule	**HANDLE** Identifies the loaded library module.

Return Value None.

FreeModule 3.0

Syntax void **FreeModule**(*hModule*)

This function decreases the reference count of the loaded module by one. When the reference count reaches zero, the memory occupied by the module is freed.

Parameter	Type/Description
hModule	**HANDLE** Identifies the loaded module.

Return Value None.

FreeProcInstance

Syntax **void FreeProcInstance**(*lpProc*)

This function frees the function specified by the *lpProc* parameter from the data segment bound to it by the **MakeProcInstance** function.

Parameter	Type/Description
lpProc	**FARPROC** Is the procedure-instance address of the function to be freed. It must have been created previously by using the **Make-ProcInstance** function.

Return Value None.

Comments After freeing a procedure instance, attempts to call the function using the freed procedure-instance address will result in an unrecoverable error.

FreeResource

Syntax **BOOL FreeResource**(*hResData*)

This function removes a loaded resource from memory by freeing the allocated memory occupied by that resource.

The **FreeResource** function does not actually free the resource until the reference count is zero (that is, the number of calls to the function equals the number of times the application called the **LoadResource** function for this resource). This ensures that the data remain in memory for the application to use.

Parameter	Type/Description
hResData	**HANDLE** Identifies the data associated with the resource. The handle is assumed to have been created by using the **LoadResource** function.

Return Value The return value specifies the outcome of the function. The return value is nonzero if the function has failed and the resource has not been freed. The return value is zero if the function is successful.

FreeSelector 3.0

Syntax **WORD** **FreeSelector**(*wSelector*)

This function frees a selector originally allocated by the **AllocSelector**, **AllocCStoDS-Alias**, or **AllocDStoCSAlias** functions. After the application calls this function, the selector is invalid and must not be used.

Parameter	Type/Description
wSelector	**WORD** Specifies the selector to be freed.

Return Value The return value is NULL if the function was successful. Otherwise, it is the selector specified by the *wSelector* parameter.

Comments An application should not use this function unless it is absolutely necessary. Use of this function violates preferred Windows programming practices.

GetActiveWindow

Syntax **HWND GetActiveWindow( )**

This function retrieves the window handle of the active window. The active window is either the window that has the current input focus, or the window explicitly made active by the **SetActiveWindow** function.

This function has no parameters.

Return Value The return value identifies the active window.

GetAspectRatioFilter

Syntax **DWORD GetAspectRatioFilter**(*hDC*)

This function retrieves the setting for the current aspect-ratio filter. The aspect ratio is the ratio formed by a device's pixel width and height. Information about a device's aspect ratio is used in the creation, selection, and displaying of fonts. Windows provides a special filter, the aspect-ratio filter, to select fonts designed for a particular aspect ratio from all of the available fonts. The filter uses the aspect ratio specified by the **SetMapperFlags** function.

Parameter	Type/Description
hDC	**HDC** Identifies the device context that contains the specfied aspect ratio.

Return Value The return value specifies the aspect ratio used by the current aspect-ratio filter. The *x*-coordinate of the aspect ratio is contained in the high-order word, and the *y*-coordinate is contained in the low-order word.

GetAsyncKeyState

Syntax **int GetAsyncKeyState**(*vKey*)

This function determines whether a key is up or down at the time the function is called, and whether the key was pressed after a previous call to the **GetAsyncKeyState** function. If the most significant bit of the return value is set, the key is currently down; if the least significant bit is set, the key was pressed after a previous call to the function.

Parameter	Type/Description
vKey	**int** Specifies one of 256 possible virtual-key code values.

Return Value The return value specifies whether the key was pressed since the last call to **GetAsyncKeyState** and whether the key is currently up or down. If the most significant bit is set, the key is down, and if the least significant bit is set, the key was pressed after a preceding **GetAsyncKeyState** call.

GetAtomHandle

Syntax **HMEM GetAtomHandle**(*wAtom*)

This function retrieves a handle (relative to the local heap) of the string that corresponds to the atom specified by the *wAtom* parameter.

Parameter	Type/Description
wAtom	**WORD** Specifies an unsigned integer that identifies the atom whose handle is to be retrieved.

Return Value The return value identifies the given atom's string. It is zero if no such atom exists.

GetAtomName

Syntax **WORD GetAtomName**(*nAtom*, *lpBuffer*, *nSize*)

This function retrieves a copy of the character string associated with the *nAtom* parameter and places it in the buffer pointed to by the *lpBuffer* parameter. The *nSize* parameter specifies the maximum size of the buffer.

Parameter	Type/Description
nAtom	**ATOM** Identifies the character string to be retrieved.
lpBuffer	**LPSTR** Points to the buffer that is to receive the character string.
nSize	**int** Specifies the maximum size (in bytes) of the buffer.

Return Value The return value specifies the actual number of bytes copied to the buffer. It is zero if the specified atom is not valid.

GetBitmapBits

Syntax **DWORD GetBitmapBits**(*hBitmap*, *dwCount*, *lpBits*)

This function copies the bits of the specified bitmap into the buffer that is pointed to by the *lpBits* parameter. The *dwCount* parameter specifies the number of bytes to be copied to the buffer. The **GetObject** function should be used to determine the correct *dwCount* value for the given bitmap.

Parameter	Type/Description
hBitmap	**HBITMAP** Identifies the bitmap.
dwCount	**DWORD** Specifies the number of bytes to be copied.
lpBits	**LPSTR** Long pointer to the buffer that is to receive the bitmap. The bitmap is an array of bytes. The bitmap byte array conforms to a structure where horizontal scan lines are multiples of 16 bits.

Return Value The return value specifies the actual number of bytes in the bitmap. It is zero if there is an error.

GetBitmapDimension

Syntax **DWORD GetBitmapDimension**(*hBitmap*)

This function returns the width and height of the bitmap specified by the *hBitmap* parameter. The height and width is assumed to have been set previously by using the **SetBitmapDimension** function.

Parameter	Type/Description
hBitmap	**HBITMAP** Identifies the bitmap.

Return Value The return value specifies the width and height of the bitmap, measured in tenths of millimeters. The height is in the high-order word, and the width is in the low-order word. If the bitmap width and height have not been set by using **SetBitmapDimension**, the return value is zero.

GetBkColor

Syntax **DWORD GetBkColor**(*hDC*)

This function returns the current background color of the specified device.

Parameter	Type/Description
hDC	**HDC** Identifies the device context.

Return Value The return value specifies an RGB color value that names the current background color.

GetBkMode

Syntax **int GetBkMode**(*hDC*)

This function returns the background mode of the specified device. The background mode is used with text, hatched brushes, and pen style that is not a solid line.

Parameter	Type/Description
hDC	**HDC** Identifies the device context.

Return Value The return value specifies the current background mode. It can be OPAQUE or TRANSPARENT.

GetBrushOrg

Syntax **DWORD GetBrushOrg**(*hDC*)

This function retrieves the current brush origin for the given device context.

Parameter	Type/Description
hDC	**HDC** Identifies the device context.

Return Value The return value specifies the current origin of the brush. The *x*-coordinate is in the low word, and the *y*-coordinate is in the high word. The coordinates are assumed to be in device units.

Comments The initial brush origin is at the coordinate (0,0).

GetBValue

Syntax

BYTE GetBValue(*rgbColor*)

This macro extracts the blue value from an RGB color value.

Parameter	Type/Description
rgbColor	**DWORD** Specifies a red, a green, and a blue color field, each specifying the intensity of the given color.

Return Value

The return value specifies a byte that contains the blue value of the *rgbColor* parameter.

Comments

The value 0FFH corresponds to the maximum intensity value for a single byte; 000H corresponds to the minimum intensity value for a single byte.

GetCapture

Syntax

HWND GetCapture( )

This function retrieves a handle that identifies the window that has the mouse capture. Only one window has the mouse capture at any given time; this window receives mouse input whether or not the cursor is within its borders.

This function has no parameters.

Return Value

The return value identifies the window that has the mouse capture; it is NULL if no window has the mouse capture.

Comments

A window receives the mouse capture when its handle is passed as the *hWnd* parameter of the **SetCapture** function.

GetCaretBlinkTime

Syntax

WORD GetCaretBlinkTime( )

This function retrieves the caret blink rate. The blink rate is the elapsed time in milliseconds between flashes of the caret.

This function has no parameters.

Return Value

The return value specifies the blink rate (in milliseconds).

GetCaretPos

Syntax **void** **GetCaretPos**(*lpPoint*)

This function retrieves the caret's current position (in screen coordinates), and copies them to the **POINT** structure pointed to by the *lpPoint* parameter.

Parameter	Type/Description
lpPoint	**LPPOINT** Points to the **POINT** structure that is to receive the screen coordinates of the caret.

Return Value None.

Comments The caret position is always given in the client coordinates of the window that contains the caret.

GetCharWidth

Syntax **BOOL** **GetCharWidth**(*hDC*, *wFirstChar*, *wLastChar*, *lpBuffer*)

This function retrieves the widths of individual characters in a consecutive group of characters from the current font. For example, if the *wFirstChar* parameter identifies the letter *a* and the *wLastChar* parameter identifies the letter *z*, the **GetCharWidth** function retrieves the widths of all lowercase characters. The function stores the values in the buffer pointed to by the *lpBuffer* parameter.

Parameter	Type/Description
hDC	**HDC** Identifies the device context.
wFirstChar	**WORD** Specifies the first character in a consecutive group of characters in the current font.
wLastChar	**WORD** Specifies the last character in a consecutive group of characters in the current font.
lpBuffer	**LPINT** Points to a buffer that will receive the width values for a consecutive group of characters in the current font.

Return Value The return value specifies the outcome of the function. It is nonzero if the function is successful. Otherwise, it is zero.

Comments

If a character in the consecutive group of characters does not exist in a particular font, it will be assigned the width value of the default character.

GetClassInfo 3.0

Syntax

BOOL GetClassInfo(*hInstance*, *lpClassName*, *lpWndClass*)

This function retrieves information about a window class. The *hInstance* parameter identifies the instance of the application that created the class, and the *lpClassName* parameter identifies the window class. If the function locates the specified window class, it copies the **WNDCLASS** data used to register the window class to the **WNDCLASS** data structure pointed to by *lpWndClass*.

Parameter	Type/Description
hInstance	**HANDLE** Identifies the instance of the application that created the class. To retrieve information on classes defined by Windows (such as buttons or list boxes), set *hInstance* to NULL.
lpClassName	**LPSTR** Points to a null-terminated string that contains the name of the class to find. If the high-order word of this parameter is NULL, the low-order word is assumed to be a value returned by the **MAKEINTRESOURCE** macro used when the class was created.
lpWndClass	**LPWNDCLASS** Points to the **WNDCLASS** structure to which the function will copy the class information.

Return Value

The return value is TRUE if the function found a matching class and successfully copied the data; the return value is FALSE if the function did not find a matching class.

Comments

The **lpszClassName, lpszMenuName,** and **hInstance** fields in the **WNDCLASS** data structure are *not* returned by this function. The menu name is not stored internally and cannot be returned. The class name is already known since it is passed to this function. The **GetClassInfo** function returns all other fields with the values used when the class was registered.

GetClassLong

Syntax

LONG GetClassLong(*hWnd*, *nIndex*)

This function retrieves the long value specified by the *nIndex* parameter from the **WNDCLASS** structure of the window specified by the *hWnd* parameter.

Parameter	Type/Description
hWnd	**HWND** Identifies the window.
nIndex	**int** Specifies the byte offset of the value to be retrieved. It can also be the following value:

Value	Meaning
GCL_WNDPROC	Retrieves a long pointer to the window function.

Return Value The return value specifies the value retrieved from the **WNDCLASS** structure.

Comments To access any extra four-byte values allocated when the window-class structure was created, use a positive byte offset as the index specified by the *nIndex* parameter. The first four-byte value in the extra space is at offset zero, the next four-byte value is at offset 4, and so on.

GetClassName

Syntax **int GetClassName**(*hWnd*, *lpClassName*, *nMaxCount*)

This function retrieves the class name of the window specified by the *hWnd* parameter.

Parameter	Type/Description
hWnd	**HWND** Identifies the window whose class name is to be retrieved.
lpClassName	**LPSTR** Points to the buffer that is to receive the class name.
nMaxCount	**int** Specifies the maximum number of bytes to be stored in the *lpClassName* parameter. If the actual name is longer, a truncated name is copied to the buffer.

Return Value The return value specifies the number of characters actually copied to *lpClassName*. The return value is zero if the specified class name is not valid.

GetClassWord

Syntax

WORD GetClassWord(*hWnd*, *nIndex*)

This function retrieves the word that is specified by the *nIndex* parameter from the **WNDCLASS** structure of the window specified by the *hWnd* parameter.

Parameter	Type/Description
hWnd	**HWND** Identifies the window.
nIndex	**int** Specifies the byte offset of the value to be retrieved. It can also be one of the following values:

Value	Meaning
GCW_CBCLSEXTRA	Tells how many bytes of additional class information you have. For information on how to access this memory, see the following "Comments" section.
GCW_CBWNDEXTRA	Tells how many bytes of additional window information you have. For information on how to access this memory, see the following "Comments" section.
GCW_HBRBACKGROUND	Retrieves a handle to the background brush.
GCW_HCURSOR	Retrieves a handle to the cursor.
GCW_HICON	Retrieves a handle to the icon.
GCW_HMODULE	Retrieves a handle to the module.
GCW_STYLE	Retrieves the window-class style bits.

Return Value

The return value specifies the value retrieved from the **WNDCLASS** structure.

Comments

To access any extra two-byte values allocated when the window-class structure was created, use a positive byte offset as the index specified by the *nIndex* parameter, starting at zero for the first two-byte value in the extra space, 2 for the next two-byte value and so on.

GetClientRect

Syntax void **GetClientRect**(*hWnd*, *lpRect*)

This function copies the client coordinates of a window's client area into the data structure pointed to by the *lpRect* parameter. The client coordinates specify the upper-left and lower-right corners of the client area. Since client coordinates are relative to the upper-left corners of a window's client area, the coordinates of the upper-left corner are (0,0).

Parameter	Type/Description
hWnd	**HWND** Identifies the window associated with the client area.
lpRect	**LPRECT** Points to a **RECT** data structure.

Return Value None.

GetClipboardData

Syntax HANDLE **GetClipboardData**(*wFormat*)

This function retrieves data from the clipboard in the format given by the *wFormat* parameter. The clipboard must have been opened previously.

Parameter	Type/Description
wFormat	**WORD** Specifies a data format. For a description of the data formats, see the **SetClipboardData** function, later in this chapter.

Return Value The return value identifies the memory block that contains the data from the clipboard. The handle type depends on the type of data specified by the *wFormat* parameter. It is NULL if there is an error.

Comments The available formats can be enumerated in advance by using the **EnumClipboardFormats** function.

The data handle returned by **GetClipboardData** is controlled by the clipboard, not by the application. The application should copy the data immediately, instead of relying on the data handle for long-term use. The application should not free the data handle or leave it locked.

Windows supports two formats for text, CF_TEXT and CF_OEMTEXT. CF_TEXT is the default Windows text clipboard format, while Windows uses the CF_OEMTEXT format for text in non-Windows applications. If you call **GetClipboardData** to retrieve data in one text format and the other text format is the only available text format, Windows automatically converts the text to the requested format before supplying it to your application.

If the clipboard contains data in the CF_PALETTE (logical color palette) format, the application should assume that any other data in the clipboard is realized against that logical palette.

GetClipboardFormatName

Syntax **int GetClipboardFormatName(***wFormat*, *lpFormatName*, *nMaxCount***)**

This function retrieves from the clipboard the name of the registered format specified by the *wFormat* parameter. The name is copied to the buffer pointed to by the *lpFormatName* parameter.

Parameter	Type/Description
wFormat	**WORD** Specifies the type of format to be retrieved. It must not specify any of the predefined clipboard formats.
lpFormatName	**LPSTR** Points to the buffer that is to receive the format name.
nMaxCount	**int** Specifies the maximum length (in bytes) of the string to be copied to the buffer. If the actual name is longer, it is truncated.

Return Value The return value specifies the actual length of the string copied to the buffer. It is zero if the requested format does not exist or is a predefined format.

GetClipboardOwner

Syntax **HWND GetClipboardOwner( )**

This function retrieves the window handle of the current owner of the clipboard.

This function has no parameters.

Return Value The return value identifies the window that owns the clipboard. It is NULL if the clipboard is not owned.

Comments The clipboard can still contain data even if the clipboard is not currently owned.

GetClipboardViewer

Syntax **HWND GetClipboardViewer( )**

This function retrieves the window handle of the first window in the clipboard-viewer chain.

This function has no parameters.

Return Value The return value identifies the window currently responsible for displaying the clipboard. It is NULL if there is no viewer.

GetClipBox

Syntax **int GetClipBox(***hDC, lpRect***)**

This function retrieves the dimensions of the tightest bounding rectangle around the current clipping boundary. The dimensions are copied to the buffer pointed to by the *lpRect* parameter.

Parameter	Type/Description
hDC	**HDC** Identifies the device context.
lpRect	**LPRECT** Points to the **RECT** data structure that is to receive the rectangle dimensions.

Return Value The return value specifies the clipping region's type. It can be any one of the following values:

Value	Meaning
COMPLEXREGION	Clipping region has overlapping borders.
ERROR	Device context is not valid.
NULLREGION	Clipping region is empty.
SIMPLEREGION	Clipping region has no overlapping borders.

GetCodeHandle

Syntax **HANDLE GetCodeHandle**(*lpProc*)

This function determines which code segment contains the function pointed to by the *lpProc* parameter.

Parameter	Type/Description
lpProc	**FARPROC** Is a procedure-instance address.

Return Value The return value identifies the code segment that contains the function.

Comments If the code segment that contains the function is already loaded, the **GetCodeHandle** function marks the segment as recently used. If the code segment is not loaded, **GetCodeHandle** attempts to load it. Thus, an application can use this function to attempt to preload one or more segments needed to perform a particular task.

GetCodeInfo 3.0

Syntax **void GetCodeInfo**(*lpProc*, *lpSegInfo*)

This function retrieves a pointer to an array of 16-bit values containing information about the code segment that contains the function pointed to by the *lpProc* parameter.

Parameter	Type/Description
lpProc	**FARPROC** Is the address of the function in the segment for which information is to be retrieved. Instead of a segment:offset address, this value can also be in the form of a module handle and segment number. The **GetModuleHandle** function returns the handle of a named module.
lpSegInfo	**LPVOID** Points to an array of four 32-bit values that will be filled with information about the code segment. See the following "Comments" section for a description of the values in this array.

Return Value None.

Comments The *lpSegInfo* parameter points to an array of four 32-bit values that contains such information as the location and size of the segment and its attributes. The following list describes each of these values:

Offset	Description
0	Specifies the logical-sector offset (in bytes) to the contents of the segment data, relative to the beginning of the file. Zero means no file data is available.
2	Specifies the length of the segment in the file (in bytes). Zero means 64K.
4	Contains flags which specify attributes of the segment. The following list describes these flags:

Bit	Meaning
0–2	Specifies the segment type. If bit 0 is set to 1, the segment is a data segment. Otherwise, the segment is a code segment.
3	Specifies whether segment data is iterated. When this bit set to 1, the segment data is iterated.
4	Specifies whether the segment is moveable or fixed. When this bit is set to 1, the segment is moveable. Otherwise, it is fixed.
5	Is not returned.
6	Is not returned.
7	Specifies whether the segment is a read-only data segment or an execute-only code segment. If this bit is set to 1 and the segment is a code segment, the segment is an execute-only segment. If this bit is set to zero and the segment is a data segment, it is a read-only segment.
8	Specifies whether the segment has associated relocation information. If this bit is set to 1, the segment has relocation information. Otherwise, the segment does not have relocation information.
9	Specifies whether the segment has debugging information. If this bit is set to 1, the segment has debugging information. Otherwise, the segment does not have debugging information.
10–11	Is not returned.
12–15	Is not returned.

Offset	Description
6	Specifies the total amount of memory allocated for the segment. This amount may exceed the actual size of the segment. Zero means 65,536.

GetCommError

Syntax **int** GetCommError(*nCid*, *lpStat*)

In case of a communications error, Windows locks the communications port until the error is cleared by using the **GetCommError** function. This function fills the status buffer pointed to by the *lpStat* parameter with the current status of the communication device specified by the *nCid* parameter. It also returns the error codes that have occurred since the last **GetCommError** call. If *lpStat* is NULL, only the error code is returned. For a list of the error codes, see Table 4.8, "Communications Error Codes."

Parameter	Type/Description
nCid	**int** Specifies the communication device to be examined. The **OpenComm** function returns this value.
lpStat	**COMSTAT FAR *** Points to the **COMSTAT** structure that is to receive the device status. The structure contains information about a communication device.

Return Value The return value specifies the error codes returned by the most recent communications function. It can be a combination of one or more of the values given in Table 4.8.

Table 4.8 Communications Error Codes

Value	Meaning
CE_BREAK	The hardware detects a break condition.
CE_CTSTO	Clear-to-send timeout. CTS is low for the duration specified by **CtsTimeout** while trying to transmit a character.
CE_DNS	The parallel device is not selected.
CE_DSRTO	Data-set-ready timeout. DSR is low for the duration specified by **DsrTimeout** while trying to transmit a character.
CE_FRAME	The hardware detects a framing error.
CE_IOE	An I/O error occurs while trying to communicate with a parallel device.
CE_MODE	Requested mode is not supported, or the *nCid* parameter is invalid. If set, this is the only valid error.
CE_OOP	The parallel device signals that it is out of paper.
CE_OVERRUN	A character is not read from the hardware before the next character arrives. The character is lost.
CE_PTO	Timeout occurs while trying to communicate with a parallel device.
CE_RLSDTO	Receive-line-signal-detect timeout. RLSD is low for the duration specified by **RlsdTimeout** while trying to transmit a character.

Table 4.8 **Communications Error Codes** *(continued)*

Value	Meaning
CE_RXOVER	Receive queue overflow. There is either no room in the input queue or a character is received after the **EofChar** character is received.
CE_RXPARITY	The hardware detects a parity error.
CE_TXFULL	The transmit queue is full while trying to queue a character.

GetCommEventMask

Syntax **WORD GetCommEventMask**(*nCid*, *nEvtMask*)

This function retrieves the value of the current event mask, and then clears the mask. This function must be used to prevent loss of an event.

Parameter	Type/Description
nCid	**int** Specifies the communication device to be examined. The **OpenComm** function returns this value.
nEvtMask	**int** Specifies which events are to be enabled. For a list of the event values, see the **SetCommEventMask** function, later in this chapter.

Return Value The return value specifies the current event-mask value. Each bit in the event mask specifies whether a given event has occurred. A bit is set to 1 if the event has occurred.

GetCommState

Syntax **int GetCommState**(*nCid*, *lpDCB*)

This function fills the buffer pointed to by the *lpDCB* parameter with the device control block of the communication device specified by the *nCid* parameter.

Parameter	Type/Description
nCid	**int** Specifies the device to be examined. The **OpenComm** function returns this value.
lpDCB	**DCB FAR** * Points to the **DCB** data structure that is to receive the current device control block. The structure defines the control setting for the device.

Return Value The return value specifies the outcome of the function. It is zero if the function was successful. If an error occurred, the return value is negative.

GetCurrentPDB 3.0

Syntax **WORD GetCurrentPDB()**

This function returns the paragraph address or selector of the current DOS Program Data Base (PDB), also known as the Program Segment Prefix (PSP).

This function has no parameters.

Return Value The return value is the paragraph address or selector of the current PDB.

GetCurrentPosition

Syntax **DWORD GetCurrentPosition**(*hDC*)

This function retrieves the logical coordinates of the current position.

Parameter	Type/Description
hDC	**HDC** Identifies a device context.

Return Value The return value specifies the current position. The *y*-coordinate is in the high-order word; the *x*-coordinate is in the low-order word.

GetCurrentTask

Syntax **HANDLE GetCurrentTask( )**

This function returns the handle of the currently executing task.

This function has no parameters.

Return Value The return value identifies the task if the function is successful. Otherwise, it is NULL.

GetCurrentTime

Syntax **DWORD GetCurrentTime( )**

This function retrieves the current Windows time. Windows time is the number of milliseconds that have elapsed since the system was booted.

This function has no parameters.

Return Value The return value specifies the current time (in milliseconds).

Comments The **GetCurrentTime** and **GetMessageTime** functions return different times. **GetMessageTime** returns the Windows time when the given message was created, not the current Windows time.

The system timer eventually overflows and resets to zero.

GetCursorPos

Syntax **void GetCursorPos**(*lpPoint*)

This function retrieves the cursor's current position (in screen coordinates), that copies them to the **POINT** structure pointed to by the *lpPoint* parameter.

Parameter	Type/Description
lpPoint	**LPPOINT** Points to the **POINT** structure that is to receive the screen coordinates of the cursor.

Return Value None

Comments

The cursor position is always given in screen coordinates and is not affected by the mapping mode of the window that contains the cursor.

GetDC

Syntax

HDC GetDC(*hWnd*)

This function retrieves a handle to a display context for the client area of the given window. The display context can be used in subsequent GDI functions to draw in the client area.

The **GetDC** function retrieves a common, class, or private display context depending on the class style specified for the given window. For common display contexts, **GetDC** assigns default attributes to the context each time it is retrieved. For class and private contexts, **GetDC** leaves the previously assigned attributes unchanged.

Parameter	Type/Description
hWnd	**HWND** Identifies the window whose display context is to be retrieved.

Return Value

The return value identifies the display context for the given window's client area if the function is successful. Otherwise, it is NULL.

Comments

After painting with a common display context, the **ReleaseDC** function must be called to release the context. Class and private display contexts do not have to be released. Since only five common display contexts are available at any given time, failure to release a display context can prevent other applications from accessing a display context.

GetDCOrg

Syntax

DWORD GetDCOrg(*hDC*)

This function obtains the final translation origin for the device context. The final translation origin specifies the offset used by Windows to translate device coordinates into client coordinates for points in an application's window. The final translation origin is relative to the physical origin of the screen display.

Parameter	Type/Description
hDC	**HDC** Identifies the device context whose origin is to be retrieved.

Return Value The return value specifies the final translation origin (in device coordinates). The *y*-coordinate is in the high-order word; the *x*-coordinate is in the low-order word.

GetDesktopWindow 3.0

Syntax **HWND GetDesktopWindow( )**

This function returns the window handle to the Windows desktop window. The desktop window covers the entire screen and is the area on top of which all icons and other windows are painted.

This function has no parameters.

Return Value The return value identifies the Windows desktop window.

GetDeviceCaps

Syntax **int GetDeviceCaps(***hDC***,** *nIndex***)**

This function retrieves device-specific information about a given display device. The *nIndex* parameter specifies the type of information desired.

Parameter	Type/Description
hDC	**HDC** Identifies the device context.
nIndex	**int** Specifies the item to return. It can be any one of the values given in Table 4.9, "GDI Information Indexes."

Return Value The return value specifies the value of the desired item.

Comments

Table 4.9 lists the values for the *nIndex* parameter:

Table 4.9 GDI Information Indexes

Index	Meaning
DRIVERVERSION	Version number; for example, 0x100 for 1.0.
TECHNOLOGY	Device technology. It can be any one of the following values: **Value** — **Meaning** DT_PLOTTER — Vector plotter DT_RASDISPLAY — Raster display DT_RASPRINTER — Raster printer DT_RASCAMERA — Raster camera DT_CHARSTREAM — Character stream DT_METAFILE — Metafile DT_DISPFILE — Display file
HORZSIZE	Width of the physical display (in millimeters).
VERTSIZE	Height of the physical display (in millimeters).
HORZRES	Width of the display (in pixels).
VERTRES	Height of the display (in raster lines).
LOGPIXELSX	Number of pixels per logical inch along the display width.
LOGPIXELSY	Number of pixels per logical inch along the display height.
BITSPIXEL	Number of adjacent color bits for each pixel.
PLANES	Number of color planes.
NUMBRUSHES	Number of device-specific brushes.
NUMPENS	Number of device-specific pens.
NUMFONTS	Number of device-specific fonts.
NUMCOLORS	Number of entries in the device's color table.
ASPECTX	Relative width of a device pixel as used for line drawing.
ASPECTY	Relative height of a device pixel as used for line drawing.
ASPECTXY	Diagonal width of the device pixel as used for line drawing.
PDEVICESIZE	Size of the **PDEVICE** internal data structure.
CLIPCAPS	Flag that indicates the clipping capabilities of the device. It is 1 if the device can clip to a rectangle, 0 if it cannot.
SIZEPALETTE	Number of entries in the system palette. This index is valid only if the device driver sets the RC_PALETTE bit in the RASTERCAPS index and is available only if the driver version is 3.0 or higher.

Table 4.9 GDI Information Indexes (*continued*)

Index	Meaning
NUMRESERVED	Number of reserved entries in the system palette. This index is valid only if the device driver sets the RC_PALETTE bit in the RASTERCAPS index and is available only if the driver version is 3.0 or higher.
COLORRES	Actual color resolution of the device in bits per pixel. This index is valid only if the device driver sets the RC_PALETTE bit in the RASTERCAPS index and is available only if the driver version is 3.0 or higher.
RASTERCAPS	Value that indicates the raster capabilities of the device, as shown in the following list:

Capability	Meaning
RC_BANDING	Requires banding support.
RC_BITBLT	Capable of transferring bitmaps.
RC_BITMAP64	Capable of supporting bitmaps larger than 64K.
RC_DI_BITMAP	Capable of supporting **SetDIBits** and **GetDIBits**.
RC_DIBTODEV	Capable of supporting the **SetDIBitsToDevice** function.
RC_FLOODFILL	Capable of performing flood fills.
RC_GDI20_OUTPUT	Capable of supporting Windows version 2.0 features.
RC_PALETTE	Palette-based device.
RC_SCALING	Capable of scaling.
RC_STRETCHBLT	Capable of performing the **StretchBlt** function.
RC_STRETCHDIB	Capable of performing the **StretchDIBits** function.

Index	Meaning
CURVECAPS	A bitmask that indicates the curve capabilities of the device. The bits have the following meanings:

Bit	Meaning
0	Device can do circles.
1	Device can do pie wedges.
2	Device can do chord arcs.
3	Device can do ellipses.
4	Device can do wide borders.
5	Device can do styled borders.
6	Device can do borders that are wide and styled.

Table 4.9 GDI Information Indexes *(continued)*

Index	Meaning
	7 — Device can do interiors.
	The high byte is 0.
LINECAPS	A bitmask that indicates the line capabilities of the device. The bits have the following meanings: **Bit** — **Meaning** 0 — Reserved. 1 — Device can do polyline. 2 — Reserved. 3 — Reserved. 4 — Device can do wide lines. 5 — Device can do styled lines. 6 — Device can do lines that are wide and styled. 7 — Device can do interiors. The high byte is 0.
POLYGONAL-CAPS	A bitmask that indicates the polygonal capabilities of the device. The bits have the following meanings: **Bit** — **Meaning** 0 — Device can do alternate fill polygon. 1 — Device can do rectangle. 2 — Device can do winding number fill polygon. 3 — Device can do scanline. 4 — Device can do wide borders. 5 — Device can do styled borders. 6 — Device can do borders that are wide and styled. 7 — Device can do interiors. The high byte is 0.
TEXTCAPS	A bitmask that indicates the text capabilities of the device. The bits have the following meanings: **Bit** — **Meaning** 0 — Device can do character output precision. 1 — Device can do stroke output precision. 2 — Device can do stroke clip precision.

Table 4.9 GDI Information Indexes *(continued)*

Index	Meaning	
	3	Device can do 90-degree character rotation.
	4	Device can do any character rotation.
	5	Device can do scaling independent of *X* and *Y*.
	6	Device can do doubled character for scaling.
	7	Device can do integer multiples for scaling.
	8	Device can do any multiples for exact scaling.
	9	Device can do double-weight characters.
	10	Device can do italicizing.
	11	Device can do underlining.
	12	Device can do strikeouts.
	13	Device can do raster fonts.
	14	Device can do vector fonts.
	15	Reserved. Must be returned zero.

For a list of all the available abilities, see the **LOGFONT** data structure in Chapter 7, "Data Types and Structures," in *Reference, Volume 2*.

GetDialogBaseUnits 3.0

Syntax **LONG GetDialogBaseUnits( )**

This function returns the dialog base units used by Windows when creating dialog boxes. An application should use these values to calculate the average width of characters in the system font.

This function has no parameters.

Return Value The return value specifies the dialog base units. The high-order word contains the height in pixels of the current dialog base height unit derived from the height of the system font, and the low-order word contains the width in pixels of the current dialog base width unit derived from the width of the system font.

Comments

The values returned represent dialog base units before being scaled to actual dialog units. The actual dialog unit in the *x* direction is 1/4 of the width returned by **GetDialog-BaseUnits**. The actual dialog unit in the *y* direction is 1/8 of the height returned by the function.

To determine the actual height and width in pixels of a control, given the height (x) and width (y) in dialog units and the return value (lDlgBaseUnits) from calling **GetDialog-BaseUnits**, use the following formula:

```
(x * LOWORD(lDlgBaseUnits))/4
(y * HIWORD(lDlgBaseUnits))/8
```

To avoid rounding problems, perform the multiplication before the division in case the dialog base units are not evenly divisible by four.

GetDIBits 3.0

Syntax

int GetDIBits(*hDC*, *hBitmap*, *nStartScan*, *nNumScans*, *lpBits*, *lpBitsInfo*, *wUsage*)

This function retrieves the bits of the specified bitmap and copies them, in device-independent format, into the buffer that is pointed to by the *lpBits* parameter. The *lpBitsInfo* parameter retrieves the color format for the device-independent bits.

Parameter	Type/Description
hDC	**HDC** Identifies the device context.
hBitmap	**HBITMAP** Identifies the bitmap.
nStartScan	**WORD** Specifies the first scan line in the destination bitmap to set in *lpBits*.
nNumScans	**WORD** Specifies the number of lines to be copied.
lpBits	**LPSTR** Points to a buffer that will receive the bitmap bits in device-independent format.
lpBitsInfo	**LPBITMAPINFO** Points to a **BITMAPINFO** data structure that specifies the color format and dimension for the device-independent bitmap.
wUsage	**WORD** Specifies whether the **bmiColors[ ]** fields of the *lpBits-Info* parameter are to contain explicit RGB values or indexes into the currently realized logical palette. The *wUsage* parameter must be one of the following values:

Parameter	Type/Description	
	Value	**Meaning**
	DIB_PAL_COLORS	The color table is to consist of an array of 16-bit indexes into the currently realized logical palette.
	DIB_RGB_COLORS	The color table is to contain literal RGB values.

Return Value The return value specifies the number of scan lines copied from the bitmap. It is zero if there was an error.

Comments If the *lpBits* parameter is NULL, **GetDIBits** fills in the **BITMAPINFO** data structure to which the *lpBitsInfo* parameter points, but does not retrieve bits from the bitmap.

The bitmap identified by the *hBitmap* parameter must not be selected into a device context when the application calls this function.

The origin for device-independent bitmaps is the bottom-left corner of the bitmap, not the top-left corner, which is the origin when the mapping mode is MM_TEXT.

This function also retrieves a bitmap specification formatted for Microsoft OS/2 Presentation Manager versions 1.1 and 1.2 if the *lpBitsInfo* parameter points to a **BITMAPCORE-INFO** data structure.

GetDlgCtrlID 3.0

Syntax **int GetDlgCtrlID**(*hWnd*)

This function returns the ID value of the child window identified by the *hWnd* parameter.

Parameter	Type/Description
hWnd	**HWND** Identifies the child window.

Return Value The return value is the numeric identifier of the child window if the function is successful. If the function fails, or if *hWnd* is not a valid window handle, the return value is NULL.

Comments Since top-level windows do not have an ID value, the return value of this function is invalid if the *hWnd* parameter identifies a top-level window.

GetDlgItem

Syntax

HWND GetDlgItem(*hDlg*, *nIDDlgItem*)

This function retrieves the handle of a control contained in the dialog box specified by the *hDlg* parameter.

Parameter	Type/Description
hDlg	**HWND** Identifies the dialog box that contains the control.
nIDDlgItem	**int** Specifies the integer ID of the item to be retrieved.

Return Value

The return value identifies the given control. It is NULL if no control with the integer ID given by the *nIDDlgItem* parameter exists.

Comments

The **GetDlgItem** function can be used with any parent-child window pair, not just dialog boxes. As long as the *hDlg* parameter specifies a parent window and the child window has a unique ID (as specified by the *hMenu* parameter in the **CreateWindow** function that created the child window), **GetDlgItem** returns a valid handle to the child window.

GetDlgItemInt

Syntax

WORD GetDlgItemInt(*hDlg*, *nIDDlgItem*, *lpTranslated*, *bSigned*)

This function translates the text of a control in the given dialog box into an integer value. The **GetDlgItemInt** function retrieves the text of the control identified by the *nIDDlgItem* parameter. It translates the text by stripping any extra spaces at the beginning of the text and converting decimal digits, stopping the translation when it reaches the end of the text or encounters any nonnumeric character. If the *bSigned* parameter is nonzero, **GetDlgItemInt** checks for a minus sign (–) at the beginning of the text and translates the text into a signed number. Otherwise, it creates an unsigned value.

GetDlgItemInt returns zero if the translated number is greater than 32,767 (for signed numbers) or 65,535 (for unsigned). When errors occur, such as encountering nonnumeric characters and exceeding the given maximum, **GetDlgItemInt** copies zero to the location pointed to by the *lpTranslated* parameter. If there are no errors, *lpTranslated* receives a nonzero value. If *lpTranslated* is NULL, **GetDlgItemInt** does not warn about errors. **GetDlgItemInt** sends a WM_GETTEXT message to the control.

Parameter	Type/Description
hDlg	**HWND** Identifies the dialog box.
nIDDlgItem	**int** Specifies the integer identifier of the dialog-box item to be translated.
lpTranslated	**BOOL FAR *** Points to the Boolean variable that is to receive the translated flag.
bSigned	**BOOL** Specifies whether the value to be retrieved is signed.

Return Value The return value specifies the translated value of the dialog-box item text. Since zero is a valid return value, the *lpTranslated* parameter must be used to detect errors. If a signed return value is desired, it should be cast as an **int** type.

GetDlgItemText

Syntax **int GetDlgItemText**(*hDlg*, *nIDDlgItem*, *lpString*, *nMaxCount*)

This function retrieves the caption or text associated with a control in a dialog box. The **GetDlgItemText** function copies the text to the location pointed to by the *lpString* parameter and returns a count of the number of characters it copies.

GetDlgItemText sends a WM_GETTEXT message to the control.

Parameter	Type/Description
hDlg	**HWND** Identifies the dialog box that contains the control.
nIDDlgItem	**int** Specifies the integer identifier of the dialog-box item whose caption or text is to be retrieved.
lpString	**LPSTR** Points to the buffer to receive the text.
nMaxCount	**int** Specifies the maximum length (in bytes) of the string to be copied to *lpString*. If the string is longer than *nMaxCount*, it is truncated.

Return Value The return value specifies the actual number of characters copied to the buffer. It is zero if no text is copied.

GetDOSEnvironment 3.0

Syntax **LPSTR GetDOSEnvironment( )**

This function returns a far pointer to the environment string of the currently running task. See *Microsoft MS-DOS Programmer's Reference* for more information on the format and contents of the environment string.

This function has no parameters.

Comments Unlike an application, a dynamic-link library (DLL) does not have a copy of the environment string. As a result, a library must call this function to retrieve the environment string.

GetDoubleClickTime

Syntax **WORD GetDoubleClickTime( )**

This function retrieves the current double-click time for the mouse. A double-click is a series of two clicks of the mouse button, the second occurring within a specified time after the first. The double-click time is the maximum number of milliseconds that may occur between the first and second click of a double-click.

This function has no parameters.

Return Value The return value specifies the current double-click time (in milliseconds).

GetDriveType 3.0

Syntax **WORD GetDriveType(*nDrive*)**

This function determines whether a disk drive is removeable, fixed, or remote.

Parameter	Type/Description
nDrive	**int** Specifies the drive for which the type is to be determined. Drive A: is 0, drive B: is 1, drive C: is 2, and so on.

Return Value The return value specifies the type of drive. It can be one of the following values:

Value	Meaning
DRIVE_REMOVEABLE	Disk can be removed from the drive.
DRIVE_FIXED	Disk cannot be removed from the drive.
DRIVE_REMOTE	Drive is a remote (network) drive.

The return value is zero if the function cannot determine the drive type, or 1 if the specified drive does not exist.

GetEnvironment

Syntax **int GetEnvironment**(*lpPortName*, *lpEnviron*, *nMaxCount*)

This function retrieves the current environment that is associated with the device attached to the system port specified by the *lpPortName* parameter, and copies it into the buffer specified by the *lpEnviron* parameter. The environment, maintained by GDI, contains binary data used by GDI whenever a device context is created for the device on the given port.

The function fails if there is no environment for the given port.

An application can call this function with the *lpEnviron* parameter set to NULL to determine the size of the buffer required to hold the environment. It can then allocate the buffer and call **GetEnvironment** a second time to retrieve the environment.

Parameter	Type/Description
lpPortName	**LPSTR** Points to the null-terminated character string that specifies the name of the desired port.
lpEnviron	**LPSTR** Points to the buffer that will receive the environment.
nMaxCount	**WORD** Specifies the maximum number of bytes to copy to the buffer.

Return Value The return value specifies the number of bytes copied to *lpEnviron*. If *lpEnviron* is NULL, the return value is the size in bytes of the buffer required to hold the environment. It is zero if the environment cannot be found.

Comments

The first field in the buffer pointed to by *lpEnviron* must be the same as that passed in the *lpDeviceName* parameter of the **CreateDC** function. If *lpPortName* specifies a null port (as defined in the WIN.INI file), the device name pointed to by *lpEnviron* is used to locate the desired environment.

GetFocus

Syntax

HWND GetFocus( )

This function retrieves the handle of the window that currently owns the input focus.

This function has no parameters.

Return Value

The return value identifies the window that currently owns the focus if the function is successful. Otherwise, it is NULL.

GetFreeSpace 3.0

Syntax

DWORD GetFreeSpace(*wFlags*)

This function scans the global heap and returns the number of bytes of memory currently available.

Parameter	Type/Description
wFlags	**WORD** Specifies whether to scan the heap above or below the EMS bank line in large-frame and small-frame EMS systems. If it is set to GMEM_NOT_BANKED, **GetFreeSpace** returns the amount of memory available below the line. If *wFlags* is zero, **GetFreeSpace** returns the amount is the memory available above the EMS bank line. The *wFlags* parameter is ignored for non-EMS systems.

Return Value

The return value is the amount of available memory in bytes. This memory is not necessarily contiguous; the **GlobalCompact** function returns the number of bytes in the largest block of free global memory.

GetGValue

Syntax **BYTE GetGValue**(*rgbColor*)

This macro extracts the green value from an RGB color value.

Parameter	Type/Description
rgbColor	**DWORD** Specifies a red, a green, and a blue color field, each specifying the intensity of the given color.

Return Value The return value specifies a byte that contains the green value of the *rgbColor* parameter.

Comments The value 0FFH corresponds to the maximum intensity value for a single byte; 000H corresponds to the minimum intensity value for a single byte.

GetInputState

Syntax **BOOL GetInputState( )**

This function determines whether there are mouse, keyboard, or timer events in the system queue that require processing. An event is a record that describes interrupt-level input. Mouse events occur when a user moves the mouse or clicks a mouse button. Keyboard events occur when a user presses one or more keys. Timer events occur after a specified number of clock ticks. The system queue is the location in which Windows stores mouse, keyboard, and timer events.

This function has no parameters.

Return Value The return value specifies whether mouse, keyboard or timer input occurs. It is nonzero if input is detected. Otherwise, it is zero.

GetInstanceData

Syntax **int GetInstanceData**(*hInstance*, *pData*, *nCount*)

This function copies data from a previous instance of an application into the data area of the current instance. The *hInstance* parameter specifies which instance to copy data from, *pData* specifies where to copy the data, and *nCount* specifies the number of bytes to copy.

Parameter	Type/Description
hInstance	**HANDLE** Identifies a previous call of the application.
pData	**NPSTR** Points to a buffer in the current instance.
nCount	**int** Specifies the number of bytes to copy.

Return Value The return value specifies the number of bytes actually copied.

GetKBCodePage 3.0

Syntax **int GetKBCodePage( )**

This function determines which OEM/ANSI tables are loaded by Windows.

This function has no parameters.

Return Value The return value specifies the code page currently loaded by Windows. It can be one of the following values:

Value	Meaning
437	Default (USA, used by most countries: indicates that there is no OEMANSI.BIN in the Windows directory)
850	International (OEMANSI.BIN = XLAT850.BIN)
860	Portugal (OEMANSI.BIN = XLAT860.BIN)
861	Iceland (OEMANSI.BIN = XLAT861.BIN)
863	French Canadian (OEMANSI.BIN = XLAT863.BIN)
865	Norway/Denmark (OEMANSI.BIN = XLAT865.BIN)

Comments If the file OEMANSI.BIN is in the Windows directory, Windows reads it and overwrites the OEM/ANSI translation tables in the keyboard driver.

When the user selects a language within the Setup program and the language does not use the default code page (437), Setup copies the appropriate file (such as XLATPO.BIN) to OEMANSI.BIN in the Windows system directory. If the language uses the default code page, Setup deletes OEMANSI.BIN, if it exists, from the Windows system directory.

GetKeyboardState

Syntax

void GetKeyboardState(*lpKeyState*)

This function copies the status of the 256 virtual-keyboard keys to the buffer specified by the *lpKeyState* parameter. The high bit of each byte is set to 1 if the key is down, or it is set to 0 if it is up. The low bit is set to 1 if the key was pressed an odd number of times since startup. Otherwise, it is set to 0.

Parameter	Type/Description	
lpKeyState	**BYTE FAR ***	Points to the 256-byte buffer of virtual-key codes.

Return Value

None.

Comments

An application calls the **GetKeyboardState** function in response to a keyboard-input message. This function retrieves the state of the keyboard when the input message was generated.

To obtain state information for individual keys, follow these steps:

1. Create an array of characters that is 265 bytes long.
2. Copy the contents of the buffer pointed to by the *lpKeyState* parameter into the array.
3. Use the virtual-key code from Appendix A, "Virtual-Key Codes," in *Reference*, *Volume 2*, to obtain an individual key state.

GetKeyboardType 3.0

Syntax **int GetKeyboardType**(*nTypeFlag*)

This function retrieves the system-keyboard type.

Parameter	Type/Description
nTypeFlag	**int** Determines whether the function returns a value indicating the type or subtype of the keyboard. It may be one of the following values:

Value	Meaning
0	Function returns the keyboard type.
1	Function returns the keyboard subtype.
2	Function returns the number of function keys on the keyboard.

Return Value The return value indicates the type or subtype of the system keyboard or the number of function keys on the keyboard. The subtype is an OEM-dependent value. The type may be one of the following values:

Value	Meaning
1	IBM® PC/XT™, or compatible (83-key) keyboard
2	Olivetti® M24 "ICO" (102-key) keyboard
3	IBM AT® (84-key) or similar keyboard
4	IBM Enhanced (101- or 102-key) keyboard
5	Nokia 1050 and similar keyboards
6	Nokia 9140 and similar keyboards

The return value is zero if the *nTypeFlag* parameter is greater than 2 or if the function fails.

Comments An application can determine the number of function keys on a keyboard from the keyboard type. The following shows the number of function keys for each keyboard type:

Type	Number of Function Keys
1	10
2	12 (sometimes 18)
3	10
4	12
5	10
6	24

GetKeyNameText 3.0

Syntax **int GetKeyNameText**(*lParam*, *lpBuffer*, *nSize*)

This function retrieves a string which contains the name of a key.

The keyboard driver maintains a list of names in the form of character strings for keys with names longer than a single character. The key name is translated according to the layout of the currently installed keyboard. The translation is performed for the principal language supported by the keyboard driver.

Parameter	Type/Description
lParam	**DWORD** Specifies the 32-bit parameter of the keyboard message (such as WM_KEYDOWN) which the function is processing. Byte 3 (bits 16–23) of the long parameter is a scan code. Bit 20 is the extended bit that distinguishes some keys on an enhanced keyboard. Bit 21 is a "don't care" bit; the application calling this function sets this bit to indicate that the function should not distinguish between left and right control and shift keys, for example.
lpBuffer	**LPSTR** Specifies a buffer to receive the key name.
nSize	**WORD** Specifies the maximum length in bytes of the key name, not including the terminating NULL character.

Return Value The return value is the actual length of the string copied to *lpBuffer*.

GetKeyState

Syntax **int GetKeyState**(*nVirtKey*)

This function retrieves the state of the virtual key specified by the *nVirtKey* parameter. The state specifies whether the key is up, down, or toggled.

Parameter	Type/Description
nVirtKey	**int** Specifies a virtual key. If the desired virtual key is a letter or digit (A through Z, a through z, or 0 through 9), *nVirtKey* must be set to the ASCII value of that character. For other keys, it must be one of the values listed in Appendix A, "Virtual-Key Codes," in *Reference, Volume 2*.

Return Value The return value specifies the state of the given virtual key. If the high-order bit is 1, the key is down. Otherwise, it is up. If the low-order bit is 1, the key is toggled. A toggle key, such as the CAPSLOCK key, is toggled if it has been pressed an odd number of times since the system was started. The key is untoggled if the low bit is 0.

Comments An application calls the **GetKeyState** function in response to a keyboard-input message. This function retrieves the state of the key when the input message was generated.

G

GetLastActivePopup 3.0

Syntax **HWND GetLastActivePopup**(*hwndOwner*)

This function determines which pop-up window owned by the window identified by the *hwndOwner* parameter was most recently active.

Parameter	Type/Description
hwndOwner	**HWND** Identifies the owner window.

Return Value The return value identifies the most-recently active pop-up window. The return value will be *hwndOwner* if any of the following conditions are met:

- The window identified by *hwndOwner* itself was most recently active.
- The window identified by *hwndOwner* does not own any pop-up windows.
- The window identified by *hwndOwner* is not a top-level window or is owned by another window.

GetMapMode

Syntax **int GetMapMode**(*hDC*)

This function retrieves the current mapping mode. See the **SetMapMode** function, later in this chapter, for a description of the mapping modes.

Parameter	Type/Description
hDC	**HDC** Identifies the device context.

Return Value The return value specifies the mapping mode.

GetMenu

Syntax **HMENU GetMenu**(*hWnd*)

This function retrieves a handle to the menu of the specified window.

Parameter	Type/Description
hWnd	**HWND** Identifies the window whose menu is to be examined.

Return Value The return value identifies the menu. It is NULL if the given window has no menu. The return value is undefined if the window is a child window.

GetMenuCheckMarkDimensions 3.0

Syntax **DWORD GetMenuCheckMarkDimensions**( )

This function returns the dimensions of the default checkmark bitmap. Windows displays this bitmap next to checked menu items. Before calling the **SetMenuItemBitmaps** function to replace the default checkmark, an application should call the **GetMenuCheckMarkDimensions** function to determine the correct size for the bitmaps.

This function has no parameters.

Return Value The return value specifies the height and width of the default checkmark bitmap. The high-order word contains the height in pixels and the low-order word contains the width.

GetMenuItemCount

Syntax **WORD GetMenuItemCount**(*hMenu*)

This function determines the number of items in the menu identified by the *hMenu* parameter. This may be either a pop-up or a top-level menu.

Parameter	Type/Description
hMenu	**HMENU** Identifies the handle to the menu to be examined.

Return Value The return value specifies the number of items in the menu specified by the *hMenu* parameter if the function is successful. Otherwise, it is –1.

GetMenuItemID

Syntax **WORD GetMenuItemID**(*hMenu, nPos*)

This function obtains the menu-item identifier for a menu item located at the position defined by the *nPos* parameter.

Parameter	Type/Description
hMenu	**HMENU** Identifies a handle to the pop-up menu that contains the item whose ID is being retrieved.
nPos	**int** Specifies the position (zero-based) of the menu item whose ID is being retrieved.

Return Value The return value specifies the item ID for the specified item in a pop-up menu if the function is successful; if *hMenu* is NULL or if the specified item is a pop-up menu (as opposed to an item within the pop-up menu), the return value is –1.

GetMenuState

Syntax **WORD GetMenuState**(*hMenu, wId, wFlags*)

This function obtains the number of items in the pop-up menu associated with the menu item specified by the *wId* parameter if the *hMenu* parameter identifies a menu with an associated pop-up menu. If *hMenu* identifies a pop-up menu, this function obtains the status of the menu item associated with *wId*.

Parameter	Type/Description
hMenu	**HMENU** Identifies the menu.
wId	**WORD** Specifies the menu-item ID.
wFlags	**WORD** Specifies the nature of the *wId* parameter. If the *wFlags* parameter contains MF_BYPOSITION, *wId* specifies a (zero-based) relative position; if *wFlags* contains MF_BYCOMMAND, *wId* specifies the item ID.

Return Value

The return value specifies the outcome of the function. It is –1 if the specified item does not exist. If the menu itself does not exist, a fatal exit occurs. If *wId* identifies a pop-up menu, the return value contains the number of items in the pop-up menu in its high-order byte, and the menu flags associated with the pop-up menu in its low-order byte; otherwise, it is a mask (Boolean OR) of the values from the following list (this mask describes the status of the menu item that *wId* identifies):

Value	Meaning
MF_CHECKED	Checkmark is placed next to item (pop-up menus only).
MF_DISABLED	Item is disabled.
MF_ENABLED	Item is enabled.
MF_GRAYED	Item is disabled and grayed.
MF_MENUBARBREAK	Same as MF_MENUBREAK, except for pop-up menus where the new column is separated from the old column by a vertical dividing line.
MF_MENUBREAK	Item is placed on a new line (static menus) or in a new column (pop-up menus) without separating columns.
MF_SEPARATOR	Horizontal dividing line is drawn (pop-up menus only). This line cannot be enabled, checked, grayed, or highlighted. The *lpNewItem* and *wIDNewItem* parameters are ignored.
MF_UNCHECKED	Checkmark is not placed next to item (default).

GetMenuString

Syntax **int** **GetMenuString**(*hMenu*, *wIDItem*, *lpString*, *nMaxCount*, *wFlag*)

This function copies the label of the specified menu item into the *lpString* parameter.

Parameter	Type/Description
hMenu	**HMENU** Identifies the menu.
wIDItem	**WORD** Specifies the integer identifier of the menu item (from the resource file) or the offset of the menu item in the menu, depending on the value of the *wFlag* parameter.
lpString	**LPSTR** Points to the buffer that is to receive the label.
nMaxCount	**int** Specifies the maximum length of the label to be copied. If the label is longer than the maximum specified in *nMaxCount*, the extra characters are truncated.
wFlag	**WORD** Specifies the nature of the *wID* parameter. If *wFlags* contains MF_BYPOSITION, *wId* specifies a (zero-based) relative position; if the *wFlags* parameter contains MF_BYCOMMAND, *wId* specifies the item ID.

Return Value The return value specifies the actual number of bytes copied to the buffer.

Comments The *nMaxCount* parameter should be one larger than the number of characters in the label to accommodate the null character that terminates a string.

GetMessage

Syntax **BOOL GetMessage**(*lpMsg*, *hWnd*, *wMsgFilterMin*, *wMsgFilterMax*)

This function retrieves a message from the application queue and places the message in the data structure pointed to by the *lpMsg* parameter. If no message is available, the **GetMessage** function yields control to other applications until a message becomes available.

GetMessage retrieves only messages associated with the window specified by the *hWnd* parameter and within the range of message values given by the *wMsgFilterMin* and *wMsgFilterMax* parameters. If *hWnd* is NULL, **GetMessage** retrieves messages for any window that belongs to the application making the call. (The **GetMessage** function does not retrieve messages for windows that belong to other applications.) If *wMsgFilterMin* and *wMsgFilterMax* are both zero, **GetMessage** returns all available messages (no filtering is performed).

The constants WM_KEYFIRST and WM_KEYLAST can be used as filter values to retrieve all messages related to keyboard input; the constants WM_MOUSEFIRST and WM_MOUSELAST can be used to retrieve all mouse-related messages.

Parameter	Type/Description
lpMsg	**LPMSG** Points to an **MSG** data structure that contains message information from the Windows application queue.
hWnd	**HWND** Identifies the window whose messages are to be examined. If *hWnd* is NULL, **GetMessage** retrieves messages for any window that belongs to the application making the call.
wMsgFilterMin	**WORD** Specifies the integer value of the lowest message value to be retrieved.
wMsgFilterMax	**WORD** Specifies the integer value of the highest message value to be retrieved.

Return Value The return value specifies the outcome of the function. It is nonzero if a message other than WM_QUIT is retrieved. It is zero if the WM_QUIT message is retrieved.

The return value is usually used to decide whether to terminate the application's main loop and exit the program.

Comments In addition to yielding control to other applications when no messages are available, the **GetMessage** and **PeekMessage** functions also yield control when WM_PAINT or WM_TIMER messages for other tasks are available.

The **GetMessage**, **PeekMessage**, and **WaitMessage** functions are the only ways to let other applications run. If your application does not call any of these functions for long periods of time, other applications cannot run.

When **GetMessage**, **PeekMessage**, and **WaitMessage** yield control to other applications, the stack and data segments of the application calling the function may move in memory to accommodate the changing memory requirements of other applications. If the application has stored long pointers to objects in the data or stack segment (that is, global or local variables), these pointers can become invalid after a call to **GetMessage**, **PeekMessage**, or **WaitMessage**. The *lpMsg* parameter of the called function remains valid in any case.

GetMessagePos

Syntax

DWORD GetMessagePos( )

This function returns a long value that represents the cursor position (in screen coordinates) when the last message obtained by the **GetMessage** function occurred.

This function has no parameters.

Return Value

The return value specifies the *x*- and *y*-coordinates of the cursor position. The *x*-coordinate is in the low-order word, and the *y*-coordinate is in the high-order word. If the return value is assigned to a variable, the **MAKEPOINT** macro can be used to obtain a **POINT** structure from the return value; the **LOWORD** or **HIWORD** macro can be used to extract the *x*- or the *y*-coordinate.

G

Comments

To obtain the current position of the cursor instead of the position when the last message occurred, use the **GetCursorPos** function.

GetMessageTime

Syntax

DWORD GetMessageTime( )

This function returns the message time for the last message retrieved by the **GetMessage** function. The time is a long integer that specifies the elapsed time (in milliseconds) from the time the system was booted to the time the message was created (placed in the application queue).

This function has no parameters.

Return Value

The return value specifies the message time.

Comments

Do not assume that the return value is always increasing. The return value will "wrap around" to zero if the timer count exceeds the maximum value for long integers.

To calculate time delays between messages, subtract the time of the second message from the time of the first message.

GetMetaFile

Syntax **HANDLE GetMetaFile**(*lpFilename*)

This function creates a handle for the metafile named by the *lpFilename* parameter.

Parameter	Type/Description
lpFilename	**LPSTR** Points to the null-terminated character string that specifies the DOS filename of the metafile. The metafile is assumed to exist.

Return Value The return value identifies a metafile if the function is successful. Otherwise, it is NULL.

GetMetaFileBits

Syntax **HANDLE GetMetaFileBits**(*hMF*)

This function returns a handle to a global memory block that contains the specified metafile as a collection of bits. The memory block can be used to determine the size of the metafile or to save the metafile as a file. The memory block should not be modified.

Parameter	Type/Description
hMF	**HANDLE** Identifies the memory metafile.

Return Value The return value identifies the global memory block that contains the metafile. If an error occurs, the return value is NULL.

Comments The handle used as the *hMF* parameter becomes invalid when the **GetMetaFileBits** function returns, so the returned global memory handle must be used to refer to the metafile.

Memory blocks created by this function are unique to the calling application and are not shared by other applications. These blocks are automatically deleted when the application terminates.

GetModuleFileName

Syntax **int GetModuleFileName**(*hModule*, *lpFilename*, *nSize*)

This function retrieves the full pathname of the executable file from which the specified module was loaded. The function copies the null-terminated filename into the buffer pointed to by the *lpFilename* parameter.

Parameter	Type/Description
hModule	**HANDLE** Identifies the module or the instance of the module.
lpFilename	**LPSTR** Points to the buffer that is to receive the filename.
nSize	**int** Specifies the maximum number of characters to copy. If the filename is longer than the maximum number of characters specified by the *nSize* parameter, it is truncated.

Return Value The return value specifies the actual length of the string copied to the buffer.

GetModuleHandle

Syntax **HANDLE GetModuleHandle**(*lpModuleName*)

This function retrieves the module handle of the specified module.

Parameter	Type/Description
lpModuleName	**LPSTR** Points to a null-terminated character string that specifies the module.

Return Value The return value identifies the module if the function is successful. Otherwise, it is NULL.

GetModuleUsage

Syntax **int GetModuleUsage**(*hModule*)

This function returns the reference count of a specified module.

Parameter	Type/Description
hModule	**HANDLE** Identifies the module or an instance of the module.

Return Value The return value specifies the reference count of the module.

GetNearestColor

Syntax **DWORD GetNearestColor**(*hDC*, *crColor*)

This function returns the closest logical color to a specified logical color the given device can represent.

Parameter	Type/Description
hDC	**HDC** Identifies the device context.
crColor	**COLORREF** Specifies the color to be matched.

Return Value The return value specifies an RGB color value that names the solid color closest to the *crColor* value that the device can represent.

GetNearestPaletteIndex 3.0

Syntax **WORD GetNearestPaletteIndex**(*hPalette*, *crColor*)

This function returns the index of the entry in a logical palette which most closely matches a color value.

Parameter	Type/Description
hPalette	**HPALETTE** Identifies the logical palette.
crColor	**COLORREF** Specifies the color to be matched.

Return Value The return value is the index of an entry in a logical palette. The entry contains the color which most nearly matches the specified color.

GetNextDlgGroupItem

Syntax **HWND GetNextDlgGroupItem**(*hDlg*, *hCtl*, *bPrevious*)

This function searches for the next (or previous) control within a group of controls in the dialog box identified by the *hDlg* parameter. A group of controls consists of one or more controls with WS_GROUP style.

Parameter	Type/Description
hDlg	**HWND** Identifies the dialog box being searched.
hCtl	**HWND** Identifies the control in the dialog box where the search starts.
bPrevious	**BOOL** Specifies how the function is to search the group of controls in the dialog box. If the *bPrevious* parameter is zero, the function searches for the previous control in the group. If *bPrevious* is nonzero, the function searches for the next control in the group.

Return Value The return value identifies the next or previous control in the group.

Comments If the current item is the last item in the group and *bPrevious* is zero, the **GetNextDlgGroupItem** function returns the window handle of the first item in the group. If the current item is the first item in the group and *bPrevious* is nonzero, **GetNextDlgGroupItem** returns the window handle of the last item in the group.

GetNextDlgTabItem

Syntax **HWND GetNextDlgTabItem**(*hDlg*, *hCtl*, *bPrevious*)

This function obtains the handle of the first control that has the WS_TABSTOP style that precedes (or follows) the control identified by the *hCtl* parameter.

Parameter	Type/Description
hDlg	**HWND** Identifies the dialog box being searched.
hCtl	**HWND** Identifies the control to be used as a starting point for the search.
bPrevious	**BOOL** Specifies how the function is to search the dialog box. If the *bPrevious* parameter is zero, the function searches for the previous control in the dialog box. If *bPrevious* is nonzero, the function searches for the next control in the dialog box. Identifies the control to be used as a starting point for the search.

Return Value The return value identifies the previous (or next) control that has the WS_TABSTOP style set.

GetNextWindow

Syntax **HWND GetNextWindow**(*hWnd, wFlag*)

This function searches for a handle that identifies the next (or previous) window in the window-manager's list. The window-manager's list contains entries for all top-level windows, their associated child windows, and the child windows of any child windows. If the *hWnd* parameter is a handle to a top-level window, the function searches for the next (or previous) handle to a top-level window; if *hWnd* is a handle to a child window, the function searches for a handle to the next (or previous) child window.

Parameter	Type/Description
hWnd	**HWND** Identifies the current window.
wFlag	**WORD** Specifies whether the function returns a handle to the next window or to the previous window. It can be either of the following values:

Value	Meaning
GW_HWNDNEXT	The function returns a handle to the next window.
GW_HWNDPREV	The function returns a handle to the previous window.

G

Return Value The return value identifies the next (or the previous) window in the window-manager's list.

GetNumTasks

Syntax **int GetNumTasks( )**

This function returns the number of tasks currently executing in the system. A task is a unique instance of a Windows application.

This function has no parameters.

Return Value The return value specifies an integer that represents the number of tasks currently executing in the system.

GetObject

Syntax **int GetObject**(*hObject*, *nCount*, *lpObject*)

This function fills a buffer with the logical data that defines the logical object specified by the *hObject* parameter. The **GetObject** function copies the number of bytes of data specified by the *nCount* parameter to the buffer pointed to by the *lpObject* parameter. The function retrieves data structures of the **LOGPEN**, **LOGBRUSH**, **LOGFONT**, or **BITMAP** type, or an integer, depending on the logical object. The buffer must be sufficiently large to receive the data.

If *hObject* specifies a bitmap, the function returns only the width, height, and color format information of the bitmap. The actual bits must be retrieved by using the **GetBitmapBits** function.

If *hObject* specifies a logical palette, it retrieves a two-byte value that specifies the number of entries in the palette; it does not retrieve the entire **LOGPALETTE** data structure that defines the palette. To get information on palette entries, an application must call the **GetPaletteEntries** function.

Parameter	Type/Description
hObject	**HANDLE** Identifies a logical pen, brush, font, bitmap, or palette.
nCount	**int** Specifies the number of bytes to be copied to the buffer.
lpObject	**LPSTR** Points to the buffer that is to receive the information.

Return Value The return value specifies the actual number of bytes retrieved. It is zero if an error occurs.

GetPaletteEntries 3.0

Syntax **WORD GetPaletteEntries**(*hPalette*, *wStartIndex*, *wNumEntries*, *lpPaletteEntries*)

This function retrieves a range of palette entries in a logical palette.

Parameter	Type/Description
hPalette	**HPALETTE** Identifies the logical palette.
wStartIndex	**WORD** Specifies the first entry in the logical palette to be retrieved.
wNumEntries	**WORD** Specifies the number of entries in the logical palette to be retrieved.
lpPaletteEntries	**LPPALETTEENTRY** Points to an array of **PALETTEENTRY** data structures to receive the palette entries. The array must contain at least as many data structures as specified by the *wNumEntries* parameter.

Return Value The return value is the number of entries retrieved from the logical palette. It is zero if the function failed.

GetParent

Syntax **HWND GetParent**(*hWnd*)

This function retrieves the window handle of the specified window's parent window (if any).

Parameter	Type/Description
hWnd	**HWND** Identifies the window whose parent window handle is to be retrieved.

Return Value The return value identifies the parent window. It is NULL if the window has no parent window.

GetPixel

Syntax **DWORD GetPixel**(*hDC, X, Y*)

This function retrieves the RGB color value of the pixel at the point specified by the *X* and *Y* parameters. The point must be in the clipping region. If the point is not in the clipping region, the function is ignored.

Parameter	Type/Description
hDC	**HDC** Identifies the device context.
X	**int** Specifies the logical *x*-coordinate of the point to be examined.
Y	**int** Specifies the logical *y*-coordinate of the point to be examined.

Return Value The return value specifies an RGB color value for the color of the given point. It is –1 if the coordinates do not specify a point in the clipping region.

Comments Not all devices support the **GetPixel** function. For more information, see the RC_BITBLT raster capability in the **GetDeviceCaps** function, earlier in this chapter.

GetPolyFillMode

Syntax **int GetPolyFillMode**(*hDC*)

This function retrieves the current polygon-filling mode.

Parameter	Type/Description
hDC	**HDC** Identifies the device context.

Return Value The return value specifies the polygon-filling mode. It can be any one of the following values:

Value	Meaning
ALTERNATE	Alternate mode
WINDING	Winding-number mode

For a description of these modes, see the **SetPolyFillMode** function, later in this chapter.

GetPriorityClipboardFormat 3.0

Syntax **int GetPriorityClipboardFormat**(*lpPriorityList*, *nCount*)

This function returns the first clipboard format in a list for which data exist in the clipboard.

Parameter	Type/Description
lpPriorityList	**WORD FAR *** Points to an integer array that contains a list of clipboard formats in priority order. For a description of the data formats, see the **SetClipboardData** function later in this chapter.
nCount	**int** Specifies the number of entries in *lpPriorityList*. This value must not be greater than the actual number of entries in the list.

Return Value

The return value is the highest priority clipboard format in the list for which data exist. If no data exist in the clipboard, this function returns NULL. If data exist in the clipboard which did not match any format in the list, the return value is –1.

GetPrivateProfileInt 3.0

Syntax

WORD GetPrivateProfileInt(*lpApplicationName*, *lpKeyName*, *nDefault*, *lpFileName*)

This function retrieves the value of an integer key from the specified initialization file.

The function searches the file for a key that matches the name specified by the *lpKeyName* parameter under the application heading specified by the *lpApplicationName* parameter. An integer entry in the initialization file must have the following form:

```
[application name]
keyname = value
     .
     .
     .
```

Parameter	Type/Description
lpApplicationName	**LPSTR** Points to the name of a Windows application that appears in the initialization file.
lpKeyName	**LPSTR** Points to a key name that appears in the initialization file.
nDefault	**int** Specifies the default value for the given key if the key cannot be found in the initialization file.
lpFileName	**LPSTR** Points to a string that names the initialization file. If *lpFileName* does not contain a path to the file, Windows searches for the file in the Windows directory.

Return Value

The return value specifies the result of the function. The return value is zero if the value that corresponds to the specified key name is not an integer or if the integer is negative. If the value that corresponds to the key name consists of digits followed by nonnumeric characters, the function returns the value of the digits. For example, if the entry *Key-Name=102abc* is accessed, the function returns 102. If the key is not found, this function returns the default value, *nDefault*.

Comments

The **GetPrivateProfileInt** function is not case dependent, so the strings in *lpApplication-Name* and *lpKeyName* may be in any combination of uppercase and lowercase letters.

GetPrivateProfileString 3.0

Syntax

int GetPrivateProfileString(*lpApplicationName*, *lpKeyName*, *lpDefault*, *lpReturnedString*, *nSize*, *lpFileName*)

This function copies a character string from the specified initialization file into the buffer pointed to by the *lpReturnedString* parameter.

The function searches the file for a key that matches the name specified by the *lpKeyName* parameter under the application heading specified by the *lpApplicationName* parameter. If the key is found, the corresponding string is copied to the buffer. If the key does not exist, the default character string specified by the *lpDefault* parameter is copied. A string entry in the initialization file must have the following form:

```
[application name]
keyname = string
      .
      .
      .
```

If *lpKeyName* is NULL, the **GetPrivateProfileString** function enumerates all key names associated with *lpApplicationName* by filling the location pointed to by *lpReturnedString* with a list of key names (not values). Each key name in the list is terminated with a null character.

Parameter	Type/Description
lpApplicationName	**LPSTR** Points to the name of a Windows application that appears in the initialization file.
lpKeyName	**LPSTR** Points to a key name that appears in the initialization file.
lpDefault	**LPSTR** Specifies the default value for the given key if the key cannot be found in the initialization file.

Parameter	Type/Description
lpReturnedString	**LPSTR** Points to the buffer that receives the character string.
nSize	**int** Specifies the maximum number of characters (including the last null character) to be copied to the buffer.
lpFileName	**LPSTR** Points to a string that names the initialization file. If *lpFileName* does not contain a path to the file, Windows searches for the file in the Windows directory.

Return Value The return value specifies the number of characters copied to the buffer identified by the *lpReturnedString* parameter, not including the terminating null character. If the buffer is not large enough to contain the entire string and *lpKeyName* is not NULL, the return value is equal to the length specified by the *nSize* parameter. If the buffer is not large enough to contain the entire string and *lpKeyName* is NULL, the return value is equal to the length specified by the *nSize* parameter minus 2.

Comments **GetPrivateProfileString** is not case dependent, so the strings in *lpApplicationName* and *lpKeyName* may be in any combination of uppercase and lowercase letters.

5

GetProcAddress

Syntax FARPROC **GetProcAddress**(*hModule*, *lpProcName*)

This function retrieves the memory address of the function whose name is pointed to by the *lpProcName* parameter. The **GetProcAddress** function searches for the function in the module specified by the *hModule* parameter, or in the current module if *hModule* is NULL. The function must be an exported function; the module's definition file must contain an appropriate **EXPORTS** line for the function.

Parameter	Type/Description
hModule	**HANDLE** Identifies the library module that contains the function.
lpProcName	**LPSTR** Points to the function name, or contains the ordinal value of the function. If it is an ordinal value, the value must be in the low-order word and zero must be in the high-order word. The string must be a null-terminated character string.

Return Value The return value points to the function's entry point if the function is successful. Otherwise, it is NULL.

If the *lpProcName* parameter is an ordinal value and a function with the specified ordinal does not exist in the module, **GetProcAddress** can still return a non-NULL value. In cases where the function may not exist, specify the function by name rather than ordinal value.

Comments Only use **GetProcAddress** to retrieve addresses of exported functions that belong to library modules. The **MakeProcInstance** function can be used to access functions within different instances of the current module.

The spelling of the function name (pointed to by *lpProcName*) must be identical to the spelling as it appears in the source library's definition (.DEF) file. The function can be renamed in the definition file.

GetProfileInt

Syntax **WORD GetProfileInt**(*lpAppName, lpKeyName, nDefault*)

This function retrieves the value of an integer key from the Windows initialization file, WIN.INI. The function searches WIN.INI for a key that matches the name specified by the *lpKeyName* parameter under the application heading specified by the *lpAppName* parameter. An integer entry in WIN.INI must have the following form:

```
[application name]
keyname = value
.
.
.
```

Parameter	Type/Description
lpAppName	**LPSTR** Points to the name of a Windows application that appears in the Windows initialization file.
lpKeyName	**LPSTR** Points to a key name that appears in the Windows initialization file.
nDefault	**int** Specifies the default value for the given key if the key cannot be found in the Windows initialization file.

Return Value The return value specifies the result of the function. The return value is zero if the value that corresponds to the specified key name is not an integer or if the integer is negative. If the value that corresponds to the key name consists of digits followed by nonnumeric characters, the function returns the value of the digits. For example, if the entry *KeyName=102abc* is accessed, the function returns 102. If the key is not found, this function returns the default value, *nDefault*.

GetProfileString

Syntax **int GetProfileString**(*lpAppName*, *lpKeyName*, *lpDefault*, *lpReturnedString*, *nSize*)

This function copies a character string from the Windows initialization file, WIN.INI, into the buffer pointed to by the *lpReturnedString* parameter. The function searches WIN.INI for a key that matches the name specified by the *lpKeyName* parameter under the application heading specified by the *lpAppName* parameter. If the key is found, the corresponding string is copied to the buffer. If the key does not exist, the default character string specified by the *lpDefault* parameter is copied. A string entry in WIN.INI must have the following form:

```
[application name]
keyname = value
.
.
.
```

If *lpKeyName* is NULL, the **GetProfileString** function enumerates all key names associated with *lpAppName* by filling the location pointed to by *lpReturnedString* with a list of key names (not values). Each key name in the list is terminated with a null character.

Parameter	Type/Description
lpAppName	**LPSTR** Points to a null-terminated character string that names the application.
lpKeyName	**LPSTR** Points to a null-terminated character string that names a key.
lpDefault	**LPSTR** Specifies the default value for the given key if the key cannot be found in the initialization file.
lpReturnedString	**LPSTR** Points to the buffer that receives the character string.
nSize	**int** Specifies the number of characters (including the last null character) that will be copied to the buffer.

Return Value The return value specifies the number of characters copied to the buffer identified by the *lpReturnedString* parameter, not including the terminating null character. If the buffer is not large enough to contain the entire string and *lpKeyName* is not NULL, the return value is equal to the length specified by the *nSize* parameter. If the buffer is not large enough to contain the entire string and *lpKeyName* is NULL, the return value is equal to the length specified by the *nSize* parameter minus 2.

Comments **GetProfileString** is not case-dependent, so the strings in *lpAppName* and *lpKeyName* may be in any combination of uppercase and lowercase letters.

GetProp

Syntax

HANDLE GetProp(*hWnd*, *lpString*)

This function retrieves a data handle from the property list of the specified window. The character string pointed to by the *lpString* parameter identifies the handle to be retrieved. The string and handle are assumed to have been added to the property list by using the **SetProp** function.

Parameter	Type/Description
hWnd	**HWND** Identifies the window whose property list is to be searched.
lpString	**LPSTR** Points to a null-terminated character string or an atom that identifies a string. If an atom is given, it must have been created previously by using the **AddAtom** function. The atom, a 16-bit value, must be placed in the low-order word of the *lpString* parameter; the high-order word must be set to zero.

Return Value

The return value identifies the associated data handle if the property list contains the given string. Otherwise, it is NULL.

Comments

The value retrieved by the **GetProp** function can be any 16-bit value useful to the application.

GetRgnBox 3.0

Syntax

int GetRgnBox(*hRgn*, *lpRect*)

This function retrieves the coordinates of the bounding rectangle of the region specified by the *hRgn* parameter.

Parameter	Type/Description
hRgn	**HRGN** Identifies the region.
lpRect	**LPRECT** Points to a **RECT** data structure to receive the coordinates of the bounding rectangle.

Return Value

The return value specifies the region's type. It can be any of the following values.

Value	Meaning
COMPLEXREGION	Region has overlapping borders.
NULLREGION	Region is empty.
SIMPLEREGION	Region has no overlapping borders.

The return value is NULL if the *hRgn* parameter does not specify a valid region.

GetROP2

Syntax **int GetROP2**(*hDC*)

This function retrieves the current drawing mode. The drawing mode specifies how the pen or interior color and the color already on the display surface are combined to yield a new color.

Parameter	Type/Description
hDC	**HDC** Identifies the device context for a raster device.

Return Value The return value specifies the drawing mode. For a list of the drawing modes, see Table 4.16, "Drawing Modes," in the **SetROP2** function, later in this chapter.

Comments For more information about the drawing modes, see Chapter 11, "Binary and Ternary Raster-Operation Codes," in *Reference, Volume 2*.

GetRValue

Syntax **BYTE GetRValue**(*rgbColor*)

This macro extracts the red value from an RGB color value.

Parameter	Type/Description
rgbColor	**DWORD** Specifies a red, a green, and a blue color field, each specifying the intensity of the given color.

Return Value The return value specifies a byte that contains the red value of the *rgbColor* parameter.

Comments The value 0FFH corresponds to the maximum intensity value for a single byte; 000H corresponds to the minimum intensity value for a single byte.

GetScrollPos

Syntax **int GetScrollPos**(*hWnd*, *nBar*)

This function retrieves the current position of a scroll-bar thumb. The current position is a relative value that depends on the current scrolling range. For example, if the scrolling range is 0 to 100 and the thumb is in the middle of the bar, the current position is 50.

Parameter	Type/Description
hWnd	**HWND** Identifies a window that has standard scroll bars or a scroll-bar control, depending on the value of the *nBar* parameter.
nBar	**int** Specifies the scroll bar to examine. It can be one of the following values:

Value	Meaning
SB_CTL	Retrieves the position of a scroll-bar control. In this case, the *hWnd* parameter must be the window handle of a scroll-bar control.
SB_HORZ	Retrieves the position of a window's horizontal scroll bar.
SB_VERT	Retrieves the position of a window's vertical scroll bar.

Return Value The return value specifies the current position of the scroll-bar thumb.

GetScrollRange

Syntax **void GetScrollRange**(*hWnd*, *nBar*, *lpMinPos*, *lpMaxPos*)

This function copies the current minimum and maximum scroll-bar positions for the given scroll bar to the locations specified by the *lpMinPos* and *lpMaxPos* parameters. If the given window does not have standard scroll bars or is not a scroll-bar control, then the **GetScrollRange** function copies zero to *lpMinPos* and *lpMaxPos*.

Parameter	Type/Description
hWnd	**HWND** Identifies a window that has standard scroll bars or a scroll-bar control, depending on the value of the *nBar* parameter.
nBar	**int** Specifies an integer value that identifies which scroll bar to retrieve. It can be one of the following values:

Value	Meaning
SB_CTL	Retrieves the position of a scroll-bar control; in this case, the *hWnd* parameter must be the handle of a scroll-bar control.
SB_HORZ	Retrieves the position of a window's horizontal scroll bar.
SB_VERT	Retrieves the position of a window's vertical scroll bar.

Parameter	Type/Description
lpMinPos	**LPINT** Points to the integer variable that is to receive the minimum position.
lpMaxPos	**LPINT** Points to the integer variable that is to receive the maximum position.

Return Value None.

Comments The default range for a standard scroll bar is 0 to 100. The default range for a scroll-bar control is empty (both values are zero).

GetStockObject

Syntax HANDLE GetStockObject(*nIndex*)

This function retrieves a handle to one of the predefined stock pens, brushes, or fonts.

Parameter	Type/Description
nIndex	**int** Specifies the type of stock object desired. It can be any one of the following values:

Value	Meaning
BLACK_BRUSH	Black brush
DKGRAY_BRUSH	Dark gray brush
GRAY_BRUSH	Gray brush
HOLLOW_BRUSH	Hollow brush
LTGRAY_BRUSH	Light gray brush
NULL_BRUSH	Null brush
WHITE_BRUSH	White brush
BLACK_PEN	Black pen
NULL_PEN	Null pen
WHITE_PEN	White pen
ANSI_FIXED_FONT	ANSI fixed system font
ANSI_VAR_FONT	ANSI variable system font
DEVICE_DEFAULT_FONT	Device-dependent font
OEM_FIXED_FONT	OEM-dependent fixed font
SYSTEM_FONT	The system font. By default, Windows uses the system font to draw menus, dialog-box controls, and other text. In Windows versions 3.0 and later, the system font is proportional width; earlier versions of Windows use a fixed-width system font.

G

Parameter	Type/Description	
	Value	**Meaning**
	SYSTEM_FIXED_FONT	The fixed-width system font used in earlier versions of Windows. This stock object is available for compatibility purposes.
	DEFAULT_PALETTE	Default color palette. This palette consists of the 20 static colors always present in the system palette for matching colors in the logical palettes of background windows.

Return Value The return value identifies the desired logical object if the function is successful. Otherwise, it is NULL.

Comments The DKGRAY_BRUSH, GRAY_BRUSH, and LTGRAY_BRUSH objects should not be used as background brushes or for any other purpose in a window whose class does not specify CS_HREDRAW and CS_VREDRAW styles. Using a gray stock brush in such windows can lead to misalignment of brush patterns after a window is moved or sized. Stock-brush origins cannot be adjusted (for more information, see the **SetBrushOrg** function, later in this chapter).

GetStretchBltMode

Syntax **int GetStretchBltMode**(*hDC*)

This function retrieves the current stretching mode. The stretching mode defines how information is to be added or removed from bitmaps that are stretched or compressed by using the **StretchBlt** function.

Parameter	Type/Description
hDC	**HDC** Identifies the device context.

Return Value The return value specifies the current stretching mode. It can be WHITEONBLACK, BLACKONWHITE, or COLORONCOLOR. For more information, see the **SetStretchBltMode** function, later in this chapter.

GetSubMenu

Syntax **HMENU GetSubMenu**(*hMenu*, *nPos*)

This function retrieves the menu handle of a pop-up menu.

Parameter	Type/Description
hMenu	**HMENU** Identifies the menu.
nPos	**int** Specifies the position in the given menu of the pop-up menu. Position values start at zero for the first menu item. The pop-up menu's integer ID cannot be used in this function.

Return Value The return value identifies the given pop-up menu. It is NULL if no pop-up menu exists at the given position.

GetSysColor

Syntax **DWORD GetSysColor**(*nIndex*)

This function retrieves the current color of the display element specified by the *nIndex* parameter. Display elements are the various parts of a window and the Windows display that appear on the system display screen.

Parameter	Type/Description
nIndex	**int** Specifies the display element whose color is to be retrieved. For a list of the index values, see the **SetSysColor** function, later in this chapter.

Return Value The return value specifies an RGB color value that names the color of the given element.

Comments System colors for monochrome displays are usually interpreted as various shades of gray.

GetSysModalWindow

Syntax **HWND GetSysModalWindow( )**

This function returns the handle of a system-modal window, if one is present.

This function has no parameters.

Return Value The return value identifies the system-modal window, if one is present. If no such window is present, the return value is NULL.

GetSystemDirectory 3.0

Syntax **WORD GetSystemDirectory**(*lpBuffer*, *nSize*)

This function obtains the pathname of the Windows system subdirectory. The system subdirectory contains such files as Windows libraries, drivers, and font files.

Parameter	Type/Description
lpBuffer	**LPSTR** Points to the buffer that is to receive the null-terminated character string containing the pathname.
nSize	**int** Specifies the maximum size (in bytes) of the buffer. This value should be set to at least 144 to allow sufficient room in the buffer for the pathname.

Return Value The return value is the length of the string copied to *lpBuffer*, not including the terminating null character. If the return value is greater than *nSize*, the return value is the size of the buffer required to hold the pathname. The return value is zero if the function failed.

Comments The pathname retrieved by this function does not end with a backslash (\ ), unless the system directory is the root directory. For example, if the system directory is named WINDOWS\SYSTEM on drive C:, the pathname of the system subdirectory retrieved by this function is C:\WINDOWS\SYSTEM.

GetSystemMenu

Syntax HMENU **GetSystemMenu**(*hWnd*, *bRevert*)

This function allows the application to access the System menu for copying and modification.

Parameter	Type/Description
hWnd	**HWND** Identifies the window that will own a copy of the System menu.
bRevert	**BOOL** Specifies the action to be taken.

If *bRevert* is:	Description
zero	**GetSystemMenu** returns a handle to a copy of the System menu currently in use. This copy is initially identical to the System menu, but can be modified.
nonzero	**GetSystemMenu** destroys the possibly modified copy of the System menu (if there is one) that belongs to the specified window and returns a handle to the original, unmodified version of the System menu.

Return Value The return value identifies the System menu if *bRevert* is nonzero and the System menu has been modified. If *bRevert* is nonzero and the System menu has *not* been modified, the return value is NULL. If *bRevert* is zero, the return value identifies a copy of the System menu.

Comments Any window that does not use the **GetSystemMenu** function to make its own copy of the System menu receives the standard System menu.

The handle returned by the **GetSystemMenu** function can be used with the **AppendMenu**, **InsertMenu** or **ModifyMenu** functions to change the System menu. The System menu initially contains items identified with various ID values such as SC_CLOSE, SC_MOVE, and SC_SIZE. Menu items on the System menu send WM_SYSCOMMAND messages. All predefined System-menu items have ID numbers greater than 0xF000. If an application adds commands to the System menu, it should use ID numbers less than F000.

Windows automatically grays items on the standard System menu, depending on the situation. The application can carry out its own checking or graying by responding to the WM_INITMENU message, which is sent before any menu is displayed.

GetSystemMetrics

Syntax

int GetSystemMetrics(*nIndex*)

This function retrieves the system metrics. The system metrics are the widths and heights of various display elements of the Windows display. The **GetSystemMetrics** function can also return flags that indicate whether the current version is a debugging version, whether a mouse is present, or whether the meaning of the left and right mouse buttons have been exchanged.

Parameter	Type/Description
nIndex	**int** Specifies the system measurement to be retrieved. All measurements are given in pixels. The system measurement must be one of the values listed in Table 4.10, "System Metric Indexes."

Return Value

The return value specifies the requested system metric.

Comments

System metrics depend on the system display and may vary from display to display. Table 4.10 lists the system-metric values for the *nIndex* parameter:

Table 4.10 System Metric Indexes

Index	Meaning
SM_CXSCREEN	Width of screen.
SM_CYSCREEN	Height of screen.
SM_CXFRAME	Width of window frame that can be sized.
SM_CYFRAME	Height of window frame that can be sized.
SM_CXVSCROLL	Width of arrow bitmap on vertical scroll bar.
SM_CYVSCROLL	Height of arrow bitmap on vertical scroll bar.
SM_CXHSCROLL	Width of arrow bitmap on horizontal scroll bar.
SM_CYHSCROLL	Height of arrow bitmap on horizontal scroll bar.
SM_CYCAPTION	Height of caption.
SM_CXBORDER	Width of window frame that cannot be sized.
SM_CYBORDER	Height of window frame that cannot be sized.
SM_CXDLGFRAME	Width of frame when window has WS_DLGFRAME style.
SM_CYDLGFRAME	Height of frame when window has WS_DLGFRAME style.
SM_CXHTHUMB	Width of thumb box on horizontal scroll bar.
SM_CYVTHUMB	Height of thumb box on vertical scroll bar.

Table 4.10 System Metric Indexes *(continued)*

Index	Meaning
SM_CXICON	Width of icon.
SM_CYICON	Height of icon.
SM_CXCURSOR	Width of cursor.
SM_CYCURSOR	Height of cursor.
SM_CYMENU	Height of single-line menu bar.
SM_CXFULLSCREEN	Width of window client area for full-screen window.
SM_CYFULLSCREEN	Height of window client area for full-screen window (equivalent to the height of the screen minus the height of the window caption).
SM_CYKANJIWINDOW	Height of Kanji window.
SM_CXMINTRACK	Minimum tracking width of window.
SM_CYMINTRACK	Minimum tracking height of window.
SM_CXMIN	Minimum width of window.
SM_CYMIN	Minimum height of window.
SM_CXSIZE	Width of bitmaps contained in the title bar.
SM_CYSIZE	Height of bitmaps contained in the title bar.
SM_MOUSEPRESENT	Nonzero if mouse hardware installed.
SM_DEBUG	Nonzero if Windows debugging version.
SM_SWAPBUTTON	Nonzero if left and right mouse buttons swapped.

GetSystemPaletteEntries 3.0

Syntax **WORD GetSystemPaletteEntries**(*hDC*, *wStartIndex*, *wNumEntries*, *lpPaletteEntries*)

This function retrieves a range of palette entries from the system palette.

Parameter	Type/Description
hDC	**HDC** Identifies the device context.
wStartIndex	**WORD** Specifies the first entry in the system palette to be retrieved.
wNumEntries	**WORD** Specifies the number of entries in the system palette to be retrieved.

Parameter	Type/Description
lpPaletteEntries	**LPPALETTEENTRY** Points to an array of **PALETTE-ENTRY** data structures to receive the palette entries. The array must contain at least as many data structures as specified by the *wNumEntries* parameter.

Return Value The return value is the number of entries retrieved from the system palette. It is zero if the function failed.

GetSystemPaletteUse 3.0

Syntax **WORD GetSystemPaletteUse**(*hDC*)

This function determines whether an application has access to the full system palette. By default, the system palette contains 20 static colors which are not changed when an application realizes its logical palette. An application can gain access to most of these colors by calling the **SetSystemPaletteUse** function.

The device context identified by the *hDC* parameter must refer to a device that supports color palettes.

Parameter	Type/Description
hDC	**HDC** Identifies the device context.

Return Value The return value specifies the current use of the system palette. It is either of the following values:

Value	Meaning
SYSPAL_NOSTATIC	System palette contains no static colors except black and white.
SYSPAL_STATIC	System palette contains static colors which will not change when an application realizes its logical palette.

GetTabbedTextExtent 3.0

Syntax

DWORD GetTabbedTextExtent(*hDC*, *lpString*, *nCount*, *nTabPositions*, *lpnTabStopPositions*)

This function computes the width and height of the line of text pointed to by the *lpString* parameter. If the string contains one or more tab characters, the width of the string is based upon the tab stops specified by the *lpnTabStopPositions* parameter. The **GetTabbedTextExtent** function uses the currently selected font to compute the dimensions of the string. The width and height (in logical units) are computed without considering the current clipping region.

Parameter	Type/Description
hDC	**HDC** Identifies the device context.
lpString	**LPSTR** Points to a text string.
nCount	**int** Specifies the number of characters in the text string.
nTabPositions	**int** Specifies the number of tab-stop positions in the array to which the *lpnTabStopPositions* points.
lpnTabStopPositions	**LPINT** Points to an array of integers containing the tab-stop positions in pixels. The tab stops must be sorted in increasing order; back tabs are not allowed.

Return Value

The return value specifies the dimensions of the string. The height is in the high-order word; the width is in the low-order word.

Comments

Since some devices do not place characters in regular cell arrays (that is, they carry out kerning), the sum of the extents of the characters in a string may not be equal to the extent of the string.

If the *nTabPositions* parameter is zero and the *lpnTabStopPositions* parameter is NULL, tabs are expanded to eight average character widths.

If *nTabPositions* is 1, the tab stops will be separated by the distance specified by the first value in the array to which *lpnTabStopPositions* points.

If *lpnTabStopPositions* points to more than a single value, then a tab stop is set for each value in the array, up to the number specified by *nTabPositions*.

GetTempDrive

Syntax **BYTE** **GetTempDrive**(*cDriveLetter*)

This function takes a drive letter or zero and returns a letter that specifies the optimal drive for a temporary file (the disk drive that can provide the best access time during disk operations with a temporary file).

The **GetTempDrive** function returns the drive letter of a hard disk if the system has one. If the *cDriveLetter* parameter is zero, the function returns the drive letter of the current disk; if *cDriveLetter* is a letter, the function returns the letter of that drive or the letter of another available drive.

Parameter	Type/Description
cDriveLetter	**BYTE** Specifies a disk-drive letter.

Return Value The return value specifies the optimal disk drive for temporary files.

GetTempFileName

Syntax **int** **GetTempFileName**(*cDriveLetter*, *lpPrefixString*, *wUnique*, *lpTempFileName*)

This function creates a temporary filename of the following form:

drive:*path**prefixuuuu*.tmp

In this syntax line, *drive* is the drive letter specified by the *cDriveLetter* parameter; *path* is the pathname of the temporary file (either the root directory of the specified drive or the directory specified in the TEMP environment variable); *prefix* is all the letters (up to the first three) of the string pointed to by the *lpPrefixString* parameter; and *uuuu* is the hexadecimal value of the number specified by the *wUnique* parameter.

Parameter	Type/Description
cDriveLetter	**BYTE** Specifies the suggested drive for the temporary filename. If *cDriveLetter* is zero, the default drive is used.
lpPrefixString	**LPSTR** Points to a null-terminated character string to be used as the temporary filename prefix. This string must consist of characters in the OEM-defined character set.
wUnique	**WORD** Specifies an unsigned short integer.

Parameter	Type/Description
lpTempFileName	**LPSTR** Points to the buffer that is to receive the temporary filename. This string consists of characters in the OEM-defined character set. This buffer should be at least 144 bytes in length to allow sufficient room for the pathname.

Return Value The return value specifies a unique numeric value used in the temporary filename. If a nonzero value was given for the *wUnique* parameter, the return value specifies this same number.

Comments To avoid problems resulting from converting OEM character an string to an ANSI string, an application should call the **_lopen** function to create the temporary file.

The **GetTempFileName** function uses the suggested drive letter for creating the temporary filename, except in the following cases:

- If a hard disk is present, **GetTempFileName** always uses the drive letter of the first hard disk.
- Otherwise, if a TEMP environment variable is defined and its value begins with a drive letter, that drive letter is used.

If the TF_FORCEDRIVE bit of the *cDriveLetter* parameter is set, the above exceptions do not apply. The temporary filename will always be created in the current directory of the drive specified by *cDriveLetter*, regardless of the presence of a hard disk or the TEMP environment variable.

If the *wUnique* parameter is zero, **GetTempFileName** attempts to form a unique number based on the current system time. If a file with the resulting filename exists, the number is increased by one and the test for existence is repeated. This continues until a unique filename is found; **GetTempFileName** then creates a file by that name and closes it. No attempt is made to create and open the file when *wUnique* is nonzero.

GetTextAlign

Syntax **WORD GetTextAlign**(*hDC*)

This function retrieves the status of the text-alignment flags. The text-alignment flags determine how the **TextOut** and **ExtTextOut** functions align a string of text in relation to the string's starting point.

Parameter	Type/Description
hDC	**HDC** Identifies the device context.

Return Value

The return value specifies the status of the text-alignment flags. The return value is a combination of one or more of the following values:

Value	Meaning
TA_BASELINE	Specifies alignment of the *x*-axis and the baseline of the chosen font within the bounding rectangle.
TA_BOTTOM	Specifies alignment of the *x*-axis and the bottom of the bounding rectangle.
TA_CENTER	Specifies alignment of the *y*-axis and the center of the bounding rectangle.
TA_LEFT	Specifies alignment of the *y*-axis and the left side of the bounding rectangle.
TA_NOUPDATECP	Specifies that the current position is not updated.
TA_RIGHT	Specifies alignment of the *y*-axis and the right side of the bounding rectangle.
TA_TOP	Specifies alignment of the *x*-axis and the top of the bounding rectangle.
TA_UPDATECP	Specifies that the current position is updated.

Comments

The text-alignment flags are not necessarily single-bit flags and may be equal to zero. To verify that a particular flag is set in the return value of this function, build an application that will perform the following steps:

1. Apply the bitwise OR operator to the flag and its related flags.

   The following list shows the groups of related flags:

   - TA_LEFT, TA_CENTER, and TA_RIGHT
   - TA_BASELINE, TA_BOTTOM, and TA_TOP
   - TA_NOUPDATECP and TA_UPDATECP

2. Apply the bitwise AND operator to the result and the return value.
3. Test for the equality of this result and the flag.

The following example shows a method for determining which horizontal-alignment flag is set:

```
switch ((TA_LEFT | TA_RIGHT | TA_CENTER) & GetTextAlign(hDC)) {
   case TA_LEFT
   .
   .
   .
   case TA_RIGHT
   .
   .
   .
   case TA_CENTER
   .
   .
   .
   }
```

GetTextCharacterExtra

Syntax **int GetTextCharacterExtra**(*hDC*)

This function retrieves the current intercharacter spacing. The intercharacter spacing defines the extra space (in logical units) that the **TextOut** or **ExtTextOut** functions add to each character as they write a line. The spacing is used to expand lines of text.

If the current mapping mode is not MM_TEXT, the **GetTextCharacterExtra** function transforms and rounds the result to the nearest unit.

Parameter	Type/Description
hDC	**HDC** Identifies the device context.

Return Value The return value specifies the current intercharacter spacing.

GetTextColor

Syntax **DWORD GetTextColor**(*hDC*)

This function retrieves the current text color. The text color defines the foreground color of characters drawn by using the **TextOut** or **ExtTextOut** functions.

Parameter	Type/Description
hDC	**HDC** Identifies the device context.

Return Value The return value specifies the current text color as an RGB color value.

GetTextExtent

Syntax **DWORD GetTextExtent**(*hDC*, *lpString*, *nCount*)

This function computes the width and height of the line of text pointed to by the *lpString* parameter. The **GetTextExtent** function uses the currently selected font to compute the dimensions of the string. The width and height (in logical units) are computed without considering the current clipping region.

Parameter	Type/Description
hDC	**HDC** Identifies the device context.
lpString	**LPSTR** Points to a text string.
nCount	**int** Specifies the number of characters in the text string.

Return Value The return value specifies the dimensions of the string. The height is in the high-order word; the width is in the low-order word.

Comments Since some devices do not place characters in regular cell arrays (that is, they carry out kerning), the sum of the extents of the characters in a string may not be equal to the extent of the string.

GetTextFace

Syntax **int GetTextFace**(*hDC*, *nCount*, *lpFacename*)

This function copies the typeface name of the selected font into a buffer pointed to by the *lpFacename* parameter. The typeface name is copied as a null-terminated character string. The *nCount* parameter specifies the maximum number of characters to be copied. If the name is longer than the number of characters specified by *nCount*, it is truncated.

Parameter	Type/Description
hDC	**HDC** Identifies the device context.
nCount	**int** Specifies the size of the buffer in bytes.
lpFacename	**LPSTR** Points to the buffer that is to receive the typeface name.

Return Value The return value specifies the actual number of bytes copied to the buffer. It is zero if an error occurs.

GetTextMetrics

Syntax **BOOL GetTextMetrics**(*hDC*, *lpMetrics*)

This function fills the buffer pointed to by the *lpMetrics* parameter with the metrics for the selected font.

Parameter	Type/Description
hDC	**HDC** Identifies the device context.
lpMetrics	**LPTEXTMETRIC** Points to the **TEXTMETRIC** data structure that is to receive the metrics.

Return Value The return value specifies the outcome of the function. It is nonzero if the function is successful. Otherwise, it is zero.

GetThresholdEvent

Syntax **LPINT GetThresholdEvent( )**

This function retrieves a flag that identifies a recent threshold event. A threshold event is any transition of a voice queue from n to $n - 1$ where n is the threshold level in notes.

This function has no parameters.

Return Value The return value points to a short integer that specifies a threshold event.

GetThresholdStatus

Syntax **int GetThresholdStatus( )**

This function retrieves the threshold-event status for each voice. Each bit in the status represents a voice. If a bit is set, the voice-queue level is currently below threshold.

The **GetThresholdStatus** function also clears the threshold-event flag.

This function has no parameters.

Return Value The return value specifies the status flags of the current threshold event.

GetTickCount

Syntax **DWORD GetTickCount( )**

This function obtains the number of milliseconds that have elapsed since the system was started.

This function has no parameters.

Return Value The return value specifies the number of milliseconds that have elapsed since the system was started.

Comments The count is accurate within ±55 milliseconds.

GetTopWindow

Syntax **HWND GetTopWindow**(*hWnd*)

This function searches for a handle to the top-level child window that belongs to the parent window associated with the *hWnd* parameter. If the window has no children, this function returns NULL.

Parameter	Type/Description
hWnd	**HWND** Identifies the parent window.

Return Value The return value identifies a handle to the top-level child window in a parent window's linked list of child windows. If no child windows exist, it is NULL.

GetUpdateRect

Syntax **BOOL GetUpdateRect**(*hWnd*, *lpRect*, *bErase*)

This function retrieves the coordinates of the smallest rectangle that completely encloses the update region of the given window. If the window was created with the CS_OWNDC style and the mapping mode is not MM_TEXT, the **GetUpdateRect** function gives the rectangle in logical coordinates. Otherwise, **GetUpdateRect** gives the rectangle in client coordinates. If there is no update region, **GetUpdateRect** makes the rectangle empty (sets all coordinates to zero).

The *bErase* parameter specifies whether **GetUpdateRect** should erase the background of the update region. If *bErase* is TRUE and the update region is not empty, the background is erased. To erase the background, **GetUpdateRect** sends a WM_ERASEBKGND message to the given window.

Parameter	Type/Description
hWnd	**HWND** Identifies the window whose update region is to be retrieved.
lpRect	**LPRECT** Points to the **RECT** data structure that is to receive the client coordinates of the enclosing rectangle.
bErase	**BOOL** Specifies whether the background in the update region is to be erased.

Return Value The return value specifies the status of the update region of the given window. It is nonzero if the update region is not empty. Otherwise, it is zero.

Comments The update rectangle retrieved by the **BeginPaint** function is identical to that retrieved by the **GetUpdateRect** function.

BeginPaint automatically validates the update region, so any call to **GetUpdateRect** made immediately after the **BeginPaint** call retrieves an empty update region.

GetUpdateRgn

Syntax **int GetUpdateRgn**(*hWnd*, *hRgn*, *fErase*)

This function copies a window's update region into a region identified by the *hRgn* parameter. The coordinates of this region are relative to the upper-left corner of the window (client coordinates).

Parameter	Type/Description
hWnd	**HWND** Identifies the window that contains the region to be updated.
hRgn	**HRGN** Identifies the update region.
fErase	**BOOL** Specifies whether or not the window background should be erased and nonclient areas of child windows should be drawn. If it is zero, no drawing is done.

Return Value The return value specifies a short-integer flag that indicates the type of resulting region. It can be any one of the following values:

Value	Meaning
COMPLEXREGION	The region has overlapping borders.
ERROR	No region was created.
NULLREGION	The region is empty.
SIMPLEREGION	The region has no overlapping borders.

Comments **BeginPaint** automatically validates the update region, so any call to **GetUpdateRgn** made immediately after the **BeginPaint** call retrieves an empty update region.

GetVersion

Syntax **WORD GetVersion( )**

This function returns the current version number of Windows.

This function has no parameters.

Return Value The return value specifies the major and minor version numbers of Windows. The high-order byte specifies the minor version (revision) number; the low-order byte specifies the major version number.

GetViewportExt

Syntax **DWORD GetViewportExt**(*hDC*)

This function retrieves the *x*- and *y*-extents of the device context's viewport.

Parameter	Type/Description
hDC	**HDC** Identifies the device context.

Return Value The return value specifies the *x*- and *y*-extents (in device units). The *y*-extent is in the high-order word; the *x*-extent is in the low-order word.

GetViewportOrg

Syntax **DWORD GetViewportOrg**(*hDC*)

This function retrieves the *x*- and *y*-coordinates of the origin of the viewport associated with the specified device context.

Parameter	Type/Description
hDC	**HDC** Identifies the device context.

Return Value The return value specifies the origin of the viewport (in device coordinates). The *y*-coordinate is in the high-order word; the *x*-coordinate is in the low-order word.

GetWindow

Syntax **HWND GetWindow**(*hWnd*, *wCmd*)

This function searches for a handle to a window from the window manager's list. The window-manager's list contains entries for all top-level windows, their associated child windows, and the child windows of any child windows. The *wCmd* parameter specifies the relationship between the window identified by the *hWnd* parameter and the window whose handle is returned.

Parameter	Type/Description
hWnd	**HWND** Identifies the original window.
wCmd	**WORD** Specifies the relationship between the original window and the returned window. It may be one of the following values:

Value	Meaning
GW_CHILD	Identifies the window's first child window.
GW_HWNDFIRST	Returns the first sibling window for a child window. Otherwise, it returns the first top-level window in the list.
GW_HWNDLAST	Returns the last sibling window for a child window. Otherwise, it returns the last top-level window in the list.
GW_HWNDNEXT	Returns the window that follows the given window on the window manager's list.
GW_HWNDPREV	Returns the previous window on the window manager's list.
GW_OWNER	Identifies the window's owner.

Return Value The return value identifies a window. It is NULL if it reaches the end of the window manager's list or if the *wCmd* parameter is invalid.

GetWindowDC

Syntax **HDC GetWindowDC**(*hWnd*)

This function retrieves the display context for the entire window, including caption bar, menus, and scroll bars. A window display context permits painting anywhere in a window, including the caption bar, menus, and scroll bars, since the origin of the context is the upper-left corner of the window instead of the client area.

GetWindowDC assigns default attributes to the display context each time it retrieves the context. Previous attributes are lost.

Parameter	Type/Description
hWnd	**HWND** Identifies the window whose display context is to be retrieved.

Return Value The return value identifies the display context for the given window if the function is successful. Otherwise, it is NULL.

Comments The **GetWindowDC** function is intended to be used for special painting effects within a window's nonclient area. Painting in nonclient areas of any window is not recommended.

The **GetSystemMetrics** function can be used to retrieve the dimensions of various parts of the nonclient area, such as the caption bar, menu, and scroll bars.

After painting is complete, the **ReleaseDC** function must be called to release the display context. Failure to release a window display context will have serious effects on painting requested by applications.

GetWindowExt

Syntax **DWORD GetWindowExt**(*hDC*)

This function retrieves the *x*- and *y*-extents of the window associated with the specified device context.

Parameter	Type/Description
hDC	**HDC** Identifies the device context.

Return Value The return value specifies the *x*- and *y*-extents (in logical units). The *y*-extent is in the high-order word; the *x*-extent is in the low-order word.

GetWindowLong

Syntax **LONG GetWindowLong**(*hWnd*, *nIndex*)

This function retrieves information about the window identified by the *hWnd* parameter.

Parameter	Type/Description
hWnd	**HWND** Identifies the window.
nIndex	**int** Specifies the byte offset of the value to be retrieved. It can also be one of the following values:

Value	Meaning
GWL_EXSTYLE	Extended window style
GWL_STYLE	Window style
GWL_WNDPROC	Long pointer to the window function

Return Value The return value specifies information about the given window.

Comments To access any extra four-byte values allocated when the window-class structure was created, use a positive byte offset as the index specified by the *nIndex* parameter, starting at zero for the first four-byte value in the extra space, 4 for the next four-byte value and so on.

GetWindowOrg

Syntax **DWORD GetWindowOrg**(*hDC*)

This function retrieves the *x*- and *y*-coordinates of the origin of the window associated with the specified device context.

Parameter	Type/Description
hDC	**HDC** Identifies the device context.

Return Value The return value specifies the origin of the window (in logical coordinates). The *y*-coordinate is in the high-order word; the *x*-coordinate is in the low-order word.

GetWindowRect

Syntax **void GetWindowRect**(*hWnd*, *lpRect*)

This function copies the dimensions of the bounding rectangle of the specified window into the structure pointed to by the *lpRect* parameter. The dimensions are given in screen coordinates, relative to the upper-left corner of the display screen, and include the caption, border, and scroll bars, if present.

Parameter	Type/Description
hWnd	**HWND** Identifies the window.
lpRect	**LPRECT** Points to a **RECT** data structure that contains the screen coordinates of the upper-left and lower-right corners of the window.

Return Value None.

GetWindowsDirectory 3.0

Syntax **WORD GetWindowsDirectory**(*lpBuffer*, *nSize*)

This function obtains the pathname of the Windows directory. The Windows directory contains such files as Windows applications, initialization files, and help files.

Parameter	Type/Description
lpBuffer	**LPSTR** Points to the buffer that is to receive the null-terminated character string containing the pathname.
nSize	**int** Specifies the maximum size (in bytes) of the buffer. This value should be set to at least 144 to allow sufficient room in the buffer for the pathname.

Return Value The return value is the length of the string copied to *lpBuffer*, not including the terminating null character. If the return value is greater than *nSize*, the return value is the size of the buffer required to hold the pathname. The return value is zero if the function failed.

Comments The pathname retrieved by this function does not end with a backslash (\ ), unless the Windows directory is the root directory. For example, if the Windows directory is named WINDOWS on drive C:, the pathname of the Windows directory retrieved by this function is C:\WINDOWS. If Windows was installed in the root directory of drive C:, the pathname retrieved by this function is C:\.

GetWindowTask

Syntax

HANDLE **GetWindowTask**(*hWnd*)

This function searches for the handle of a task associated with the *hWnd* parameter. A task is any program that executes as an independent unit. All applications are executed as tasks. Each instance of an application is a task.

Parameter	Type/Description
hWnd	**HWND** Identifies the window for which a task handle is retrieved.

Return Value

The return value identifies the task associated with a particular window.

GetWindowText

Syntax

int **GetWindowText**(*hWnd*, *lpString*, *nMaxCount*)

This function copies the given window's caption title (if it has one) into the buffer pointed to by the *lpString* parameter. If the *hWnd* parameter identifies a control, the **GetWindowText** function copies the text within the control instead of copying the caption.

Parameter	Type/Description
hWnd	**HWND** Identifies the window or control whose caption or text is to be copied.
lpString	**LPSTR** Points to the buffer that is to receive the copied string.
nMaxCount	**int** Specifies the maximum number of characters to be copied to the buffer. If the string is longer than the number of characters specified in the *nMaxCount* parameter, it is truncated.

Return Value

The return value specifies the length of the copied string. It is zero if the window has no caption or if the caption is empty.

Comments

This function causes a WM_GETTEXT message to be sent to the given window or control.

GetWindowTextLength

Syntax **int GetWindowTextLength**(*hWnd*)

This function returns the length of the given window's caption title. If the *hWnd* parameter identifies a control, the **GetWindowTextLength** function returns the length of the text within the control instead of the caption.

Parameter	Type/Description
hWnd	**HWND** Identifies the window or control.

Return Value The return value specifies the text length. It is zero if no such text exists.

GetWindowWord

Syntax **WORD GetWindowWord**(*hWnd*, *nIndex*)

This function retrieves information about the window identified by *hWnd*.

Parameter	Type/Description
hWnd	**HWND** Identifies the window.
nIndex	**int** Specifies the byte offset of the value to be retrieved. It can also be one of the following values:

Value	Meaning
GWW_HINSTANCE	Instance handle of the module that owns the window.
GWW_HWNDPARENT	Handle of the parent window, if any. The **SetParent** function changes the parent window of a child window. An application should not call the **SetWindowLong** function to change the parent of a child window.
GWW_ID	Control ID of the child window.

Return Value The return value specifies information about the given window.

Comments

To access any extra two-byte values allocated when the window-class structure was created, use a positive byte offset as the index specified by the *nIndex* parameter, starting at zero for the first two-byte value in the extra space, 2 for the next two-byte value and so on.

GetWinFlags 3.0

Syntax

DWORD GetWinFlags( )

This function returns a 32-bit value containing flags which specify the memory configuration under which Windows is running.

This function has no parameters.

Return Value

The return value contains flags specifying the current memory configuration. These flags may be any of the following values:

Value	Meaning
WF_80x87	System contains an Intel math coprocessor.
WF_CPU086	System CPU is an 8086.
WF_CPU186	System CPU is an 80186.
WF_CPU286	System CPU is an 80286.
WF_CPU386	System CPU is an 80386.
WF_CPU486	System CPU is an 80486.
WF_ENHANCED	Windows is running in 386 enhanced mode. The WF_PMODE flag is always set when WF_ENHANCED is set.
WF_LARGEFRAME	Windows is running in EMS large-frame memory configuration.
WF_PMODE	Windows is running in protected mode. This flag is always set when either WF_ENHANCED or WF_STANDARD is set.
WF_SMALLFRAME	Windows is running in EMS small-frame memory configuration.
WF_STANDARD	Windows is running in standard mode. The WF_PMODE flag is always set when WF_STANDARD is set.

If neither WF_ENHANCED nor WF_STANDARD is set, Windows is running in real mode.

GlobalAddAtom

Syntax ATOM **GlobalAddAtom**(*lpString*)

This function adds the character string pointed to by the *lpString* parameter to the atom table and creates a new global atom that uniquely identifies the string. A global atom is an atom that is available to all applications. The atom can be used in a subsequent **GlobalGetAtomName** function to retrieve the string from the atom table.

The **GlobalAddAtom** function stores no more than one copy of a given string in the atom table. If the string is already in the table, the function returns the existing atom value and increases the string's reference count by one. The string's reference count is a number that specifies the number of times **GlobalAddAtom** has been called for a particular string.

Parameter	Type/Description
lpString	**LPSTR** Points to the character string to be added to the table. The string must be a null-terminated character string.

Return Value The return value identifies the newly created atom if the function is successful. Otherwise, it is NULL.

Comments The atom values returned by **GlobalAddAtom** are within the range 0xC000 to 0xFFFF.

GlobalAlloc

Syntax HANDLE **GlobalAlloc**(*wFlags*, *dwBytes*)

This function allocates the number of bytes of memory specified by the *dwBytes* parameter from the global heap. The memory can be fixed or moveable, depending on the memory type specified by the *wFlags* parameter.

Parameter	Type/Description
wFlags	**WORD** Specifies one or more flags that tell the **GlobalAlloc** function how to allocate the memory. It can be one or more of the following values:

Value	Meaning
GMEM_DDESHARE	Allocates sharable memory. This is used for dynamic data exchange (DDE) only. Note, however, that Windows automatically discards memory allocated with this attribute when the application that allocated the memory terminates.
GMEM_DISCARDABLE	Allocates discardable memory. Can only be used with GMEM_MOVEABLE.
GMEM_FIXED	Allocates fixed memory.
GMEM_MOVEABLE	Allocates moveable memory. Cannot be used with GMEM_FIXED.
GMEM_NOCOMPACT	Does not compact or discard to satisfy the allocation request.
GMEM_NODISCARD	Does not discard to satisfy the allocation request.
GMEM_NOT_BANKED	Allocates non-banked memory. Cannot be used with GMEM_NOTIFY.
GMEM_NOTIFY	Calls the notification routine if the memory object is ever discarded.
GMEM_ZEROINIT	Initializes memory contents to zero.

Choose GMEM_FIXED or GMEM_MOVEABLE, and then combine others as needed by using the bitwise OR operator.

dwBytes	**DWORD** Specifies the number of bytes to be allocated.

Return Value The return value identifies the allocated global memory if the function is successful. Otherwise, it is NULL.

Comments

If this function is successful, it allocates at least the amount requested. The actual amount allocated may be greater, and the application can use the entire amount. To determine the actual amount allocated, call the **GlobalSize** function.

The largest block of memory that an application can allocate is 1 MB in standard mode and 64 MB in 386 enhanced mode.

GlobalCompact

Syntax

DWORD GlobalCompact(*dwMinFree*)

This function generates the number of free bytes of global memory specified by the *dwMinFree* parameter by compacting and, if necessary, discarding from the system's global heap. The function *always* compacts memory before checking for free memory. It then checks the global heap for the number of contiguous free bytes specified by the *dwMinFree* parameter. If the bytes do not exist, the **GlobalCompact** function discards unlocked discardable blocks until the requested space is generated, whenever possible.

Parameter	Type/Description
dwMinFree	**DWORD** Specifies the number of free bytes desired.

Return Value

The return value specifies the number of bytes in the largest block of free global memory.

Comments

If *dwMinFree* is zero, the return value specifies the number of bytes in the largest free segment that Windows can generate if it removes all discardable segments.

If an application uses the return value as the *dwBytes* parameter to the **GlobalAlloc** function, the GMEM_NOCOMPACT or GMEM_NODISCARD flags should not be used.

GlobalDeleteAtom

Syntax

ATOM GlobalDeleteAtom(*nAtom*)

This function decreases the reference count of a global atom by one. If the atom's reference count becomes zero, this function removes the associated string from the atom table. (A global atom is an atom that is available to all Windows applications.)

An atom's reference count specifies the number of times the atom has been added to the atom table. The **GlobalAddAtom** function increases the count on each call; the **GlobalDeleteAtom** function decreases the count on each call. **GlobalDeleteAtom** removes the string only if the atom's reference count is zero.

Parameter	Type/Description
nAtom	**ATOM** Identifies the atom and character string to be deleted.

Return Value The return value specifies the outcome of the function. It is NULL if the function is successful. It is equal to *nAtom* if the function failed and the atom has not been deleted.

GlobalDiscard

Syntax **HANDLE GlobalDiscard**(*hMem*)

This function discards a global memory block specified by the *hMem* parameter. The lock count of the memory block must be zero.

The global memory block is removed from memory, but its handle remains valid. An application can subsequently pass the handle to the **GlobalReAlloc** function to allocate another global memory block identified by the same handle.

Parameter	Type/Description
hMem	**HANDLE** Identifies the global memory block to be discarded.

Return Value The return value identifies the discarded block if the function is successful. Otherwise, it is zero.

Comments The **GlobalDiscard** function discards only global objects that an application allocated with the GMEM_DISCARDABLE and GMEM_MOVEABLE flags set. The function fails if an application attempts to discard a fixed or locked object.

GlobalDosAlloc 3.0

Syntax **DWORD GlobalDosAlloc**(*dwBytes*)

This function allocates global memory which can be accessed by DOS running in real mode. The memory is guaranteed to exist in the first megabyte of linear address space.

Parameter	Type/Description
dwBytes	**DWORD** Specifies the number of bytes to be allocated.

Return Value The return value contains a paragraph-segment value in its high-order word and a selector in its low-order word. An application can use the paragraph-segment value to access memory in real mode and the selector to access memory in protected mode. If Windows is running in real mode, the high-order and low-order words will be equal. If Windows cannot allocate a block of memory of the requested size, the return value is NULL.

Comments An application should not use this function unless it is absolutely necessary. The memory pool from which the object is allocated is a scarce system resource.

GlobalDosFree 3.0

Syntax **WORD GlobalDosFree**(*wSelector*)

This function frees a block of global memory previously allocated by a call to the **GlobalDosAlloc** function.

Parameter	Type/Description
wSelector	**WORD** Specifies the memory to be freed.

Return Value The return value identifies the outcome of the function. It is NULL if the function is successful. Otherwise, it is equal to *wSelector*.

G

GlobalFindAtom

Syntax **ATOM GlobalFindAtom**(*lpString*)

This function searches the atom table for the character string pointed to by the *lpString* parameter and retrieves the global atom associated with that string. (A global atom is an atom that is available to all Windows applications.)

Parameter	Type/Description
lpString	**LPSTR** Points to the character string to be searched for. The string must be a null-terminated character string.

Return Value The return value identifies the global atom associated with the given string. It is NULL if the string is not in the table.

GlobalFix 3.0

Syntax **void** **GlobalFix**(*hMem*)

This function prevents the global memory block identified by the *hMem* parameter from moving in linear memory. The block is locked into linear memory at its current address and its lock count is increased by one. Locked memory is not subject to moving or discarding except when the memory block is being reallocated by the **GlobalReAlloc** function. The block remains locked in memory until its lock count is decreased to zero.

Each time an application calls **GlobalFix** for a memory object, it must eventually call **GlobalUnfix** for the object. The **GlobalUnfix** function decreases the lock count for the object. Other functions also can affect the lock count of a memory object. See the description of the **GlobalFlags** function for a list of the functions that affect the lock count.

Parameter	Type/Description
hMem	**HANDLE** Identifies the global memory block.

Return Value None.

Comments Calling this function interferes with Windows memory management and results in linear-address fragmentation. Very few applications need to fix memory in linear address space.

GlobalFlags

Syntax **WORD** **GlobalFlags**(*hMem*)

This function returns information about the global memory block specified by the *hMem* parameter.

Parameter	Type/Description
hMem	**HANDLE** Identifies the global memory block.

Return Value The return value specifies a memory-allocation flag in the high byte. The flag will be one of the following values:

Value	Meaning
GMEM_DDESHARE	The block can be shared. This is used for dynamic data exchange (DDE) only.
GMEM_DISCARDABLE	The block can be discarded.
GMEM_DISCARDED	The block has been discarded.
GMEM_NOT_BANKED	The block cannot be banked.

The low byte of the return value contains the lock count of the block. Use the GMEM_LOCKCOUNT mask to retrieve the lock-count value from the return value.

Comments

To test whether or not an object can be discarded, AND the return value of **GlobalFlags** with GMEM_DISCARDABLE.

The following functions can affect the lock count of a global memory block:

Increases Lock Count	Decreases Lock Count
GlobalFix	**GlobalUnfix**
GlobalLock	**GlobalUnlock**
GlobalWire	**GlobalUnWire**
LockSegment	**UnlockSegment**

GlobalFree

Syntax

HANDLE GlobalFree(*hMem*)

This function frees the global memory block identified by the *hMem* parameter and invalidates the handle of the memory block.

Parameter	Type/Description
hMem	**HANDLE** Identifies the global memory block to be freed.

Return Value

The return value identifies the outcome of the function. It is NULL if the function is successful. Otherwise, it is equal to *hMem*.

Comments

The **GlobalFree** function must not be used to free a locked memory block, that is, a memory block with a lock count greater than zero. See the description of the **GlobalFlags** function for a list of the functions that affect the lock count.

GlobalGetAtomName

Syntax

WORD GlobalGetAtomName(*nAtom*, *lpBuffer*, *nSize*)

This function retrieves a copy of the character string associated with the *nAtom* parameter and places it in the buffer pointed to by the *lpBuffer* parameter. The *nSize* parameter specifies the maximum size of the buffer. (A global atom is an atom that is available to all Windows applications.)

Parameter	Type/Description
nAtom	**ATOM** Identifies the character string to be retrieved.
lpBuffer	**LPSTR** Points to the buffer that is to receive the character string.
nSize	**int** Specifies the maximum size (in bytes) of the buffer.

G

Return Value

The return value specifies the actual number of bytes copied to the buffer. It is zero if the specified global atom is not valid.

GlobalHandle

Syntax

DWORD GlobalHandle(*wMem*)

This function retrieves the handle of the global memory object whose segment address or selector is specified by the *wMem* parameter.

Parameter	Type/Description
wMem	**WORD** Specifies an unsigned integer value that gives the segment address or selector of a global memory object.

Return Value

The low-order word of the return value specifies the handle of the global memory object. The high-order word of the return value specifies the segment address or selector of the memory object. The return value is NULL if no handle exists for the memory object.

GlobalLock

Syntax **LPSTR** GlobalLock(*hMem*)

This function retrieves a pointer to the global memory block specified by the *hMem* parameter.

Except for nondiscardable objects in protected (standard or 386 enhanced) mode, the block is locked into memory at the given address and its lock count is increased by one. Locked memory is not subject to moving or discarding except when the memory block is being reallocated by the **GlobalReAlloc** function. The block remains locked in memory until its lock count is decreased to zero.

In protected mode, **GlobalLock** increments the lock count of discardable objects and automatic data segments only.

Each time an application calls **GlobalLock** for an object, it must eventually call **GlobalUnlock** for the object. The **GlobalUnlock** function decreases the lock count for the object if **GlobalLock** increased the lock count for the object. Other functions also can affect the lock count of a memory object. See the description of the **GlobalFlags** function for a list of the functions that affect the lock count.

Parameter	Type/Description
hMem	**HANDLE** Identifies the global memory block to be locked.

Return Value The return value points to the first byte of memory in the global block if the function is successful. If the object has been discarded or an error occurs, the return value is NULL.

Comments Discarded objects always have a lock count of zero.

GlobalLRUNewest

Syntax **HANDLE** **GlobalLRUNewest**(*hMem*)

This function moves the global memory object identified by *hMem* to the newest least-recently-used (LRU) position in memory. This greatly reduces the likelihood that the object will be discarded soon, but does not prevent the object from eventually being discarded.

Parameter	Type/Description
hMem	**HANDLE** Identifies the global memory object to be moved.

Return Value The return value is NULL if the *hMem* parameter does not specify a valid handle.

Comments This function is useful only if *hMem* is discardable.

GlobalLRUOldest

Syntax **HANDLE GlobalLRUOldest**(*hMem*)

This routine moves the global memory object identified by *hMem* to the oldest least-recently-used (LRU) position in memory and, in so doing, makes it the next candidate for discarding.

Parameter	Type/Description
hMem	**HANDLE** Identifies the global memory object to be moved.

Return Value The return value is NULL if the *hMem* parameter does not specify a valid handle.

Comments This function is useful only if *hMem* is discardable.

GlobalNotify

Syntax **void GlobalNotify**(*lpNotifyProc*)

This function installs a notification procedure for the current task. Windows calls the notification procedure whenever a global memory block allocated with the GMEM_NOTIFY flag is about to be discarded.

Parameter	Type/Description
lpNotifyProc	**FARPROC** Is the procedure instance address of the current task's notification procedure.

Return Value None.

Comments An application must not call **GlobalNotify** more than once per instance.

Windows does not call the notification procedure when it discards memory belonging to a DLL.

If the object is discarded, the application must use the GMEM_NOTIFY flag when it recreates the object by calling the **GlobalRealloc** function. Otherwise, the application will not be notified when the object is discarded again.

If the notification procedure returns a nonzero value, Windows discards the global memory block. If it returns zero, the block is not discarded.

The callback function must use the Pascal calling convention and must be declared **FAR**. The callback function must reside in a fixed code segment of a DLL.

Callback Function **BOOL FAR PASCAL** *NotifyProc*(*hMem*)

NotifyProc is a placeholder for the application-supplied function name. Export the name by including it in an EXPORTS statement in the DLL's module-definition statement.

Parameter	Type/Description	
hMem	**HANDLE**	Identifies the global memory block being discarded.

Return Value

The function returns a nonzero value if Windows is to discard the memory block, and zero if it should not.

Comments

The callback function is not necessarily called in the context of the application that owns the routine. For this reason, the callback function should not assume the stack segment of the application. The callback function should not call any routine that might move memory.

GlobalPageLock 3.0

Syntax **WORD GlobalPageLock**(*wSelector*)

This function increments the page-lock count of the memory associated with the specified global selector. As long as its page-lock count is nonzero, the data which the selector references is guaranteed to remain in memory at the same physical address and to remain paged in.

GlobalPageLock increments the page-lock count for the block of memory, and the **GlobalPageUnlock** function decrements the page-lock count. Page-locking operations can be nested, but each page lock must be balanced by a corresponding unlock.

Parameter	Type/Description
wSelector	**WORD** Specifies the selector of the memory to be page-locked.

Return Value The return value specifies the page-lock count after the function has incremented it. If the function fails, the return value is zero.

Comments An application should not use this function unless it is absolutely necessary. Use of this function violates preferred Windows programming practices. It is intended to be used for dynamically allocated data that must be accessed at interrupt time. For this reason, it must only be called from a DLL.

GlobalPageUnlock 3.0

Syntax **WORD GlobalPageUnlock**(*wSelector*)

This function decrements the page-lock count for the block of memory identified by the *wSelector* parameter and, if the page-lock count reaches zero, allows the block of memory to move and to be paged to disk.

The **GlobalPageLock** function increments the page-lock count for the block of memory, and **GlobalPageUnlock** decrements the page-lock count. Page-locking operations can be nested, but each page lock must be balanced by a corresponding unlock.

Only libraries can call this function.

Parameter	Type/Description
wSelector	**WORD** Specifies the selector of the memory to be page-unlocked.

Return Value The return value specifies the page-lock count after the function has decremented it. If the function fails, the return value is zero.

GlobalReAlloc

Syntax **HANDLE GlobalReAlloc**(*hMem*, *dwBytes*, *wFlags*)

This function reallocates the global memory block specified by the *hMem* parameter by increasing or decreasing its size to the number of bytes specified by the *dwBytes* parameter.

Parameter	Type/Description
hMem	**HANDLE** Identifies the global memory block to be reallocated.
dwBytes	**DWORD** Specifies the new size of the memory block.
wFlags	**WORD** Specifies how to reallocate the global block.

If the existing memory flags can be modified, use either one or both of the following flags (if both flags are specified, join them with the bitwise OR operator):

Value	Meaning
GMEM_DISCARDABLE	Memory can be discarded. Use only with GMEM_MODIFY.
GMEM_MODIFY	Memory flags are modified. The *dwBytes* parameter is ignored. Use only if an application will modify existing memory flags and not reallocate the memory block to a new size.
GMEM_MOVEABLE	Memory is movable. If *dwBytes* is zero, this flag causes an object previously allocated as moveable and discardable to be discarded if the block's lock count is zero. If the block is not moveable and discardable, the **GlobalReAlloc** will fail. If *dwBytes* is nonzero and the block specified by *hMem* is fixed, this flag allows the reallocated block to be moved to a new fixed location. If a moveable object is locked, this flag allows the object to be moved. This may occur even if the object is currently locked by a previous call to **GlobalLock**. (Note that the handle returned by the **GlobalReAlloc** function in this case may be different from the handle passed to the function.) Use this flag with GMEM_MODIFY to make a fixed memory block moveable.

Parameter	Type/Description	
	Value	**Meaning**
	GMEM_NOCOMPACT	Memory will not be compacted or discarded in order to satisfy the allocation request. This flag is ignored if the GMEM_MODIFY flag is set.
	GMEM_NODISCARD	Objects will not be discarded in order to satisfy the allocation request. This flag is ignored if the GMEM_MODIFY flag is set.
	GMEM_ZEROINIT	If the block is growing, the additional memory contents are initialized to zero. This flag is ignored if the GMEM_MODIFY flag is set.

Return Value

The return value identifies the reallocated global memory if the function is successful. The return value is NULL if the block cannot be reallocated.

If the function is successful, the return value is always identical to the *hMem* parameter, unless any of the following conditions is true:

- The GMEM_MOVEABLE flag is used to allow movement of a fixed block to a new fixed location.
- Windows is running in standard mode and the object is reallocated past a multiple of 65,519 bytes (17 bytes less than 64K).
- Windows is running in 386 enhanced mode and the object is reallocated past a multiple of 64K.

GlobalSize

Syntax

DWORD GlobalSize(*hMem*)

This function retrieves the current size (in bytes) of the global memory block specified by the *hMem* parameter.

Parameter	Type/Description
hMem	**HANDLE** Identifies the global memory block.

Return Value The return value specifies the actual size (in bytes) of the specified memory block. It is zero if the given handle is not valid or if the object has been discarded.

Comments The actual size of a memory block is sometimes larger than the size requested when the memory was allocated.

An application should call the **GlobalFlags** function prior to calling the **GlobalSize** function in order to verify that the specified memory block was not discarded. If the memory block were discarded, the return value for **GlobalSize** would be meaningless.

GlobalUnfix 3.0

Syntax **BOOL GlobalUnfix**(*hMem*)

This function unlocks the global memory block specified by the *hMem* parameter.

GlobalUnfix decreases the block's lock count by one. The block is completely unlocked and subject to moving or discarding if the lock count is decreased to zero. Other functions also can affect the lock count of a memory object. See the description of the **GlobalFlags** function for a list of the functions that affect the lock count.

Each time an application calls **GlobalFix** for an object, it must eventually call **GlobalUnfix** for the object.

Parameter	Type/Description
hMem	**HANDLE** Identifies the global memory block to be unlocked.

Return Value The return value specifies the outcome of the function. It is zero if the block's lock count was decreased to zero. Otherwise, the return value is nonzero.

GlobalUnlock

Syntax **BOOL GlobalUnlock**(*hMem*)

This function unlocks the global memory block specified by the *hMem* parameter.

In real mode, or if the block is discardable, **GlobalUnlock** decreases the block's lock count by one. In protected mode, **GlobalUnock** decreases the lock count of discardable objects and automatic data segments only.

The block is completely unlocked and subject to moving or discarding if the lock count is decreased to zero. Other functions also can affect the lock count of a memory object. See the description of the **GlobalFlags** function for a list of the functions that affect the lock count.

G

In all cases, each time an application calls **GlobalLock** for an object, it must eventually call **GlobalUnlock** for the object.

Parameter	Type/Description
hMem	**HANDLE** Identifies the global memory block to be unlocked.

Return Value The return value specifies the outcome of the function. It is zero if the block's lock count was decreased to zero. Otherwise, the return value is nonzero. An application should not rely on the return value to determine the number of times it must subsequently call **GlobalUnlock** for the memory block.

GlobalUnWire

Syntax **BOOL GlobalUnWire**(*hMem*)

This function unlocks a memory segment that was locked by the **GlobalWire** function and decreases the lock count by one.

The block is completely unlocked and subject to moving or discarding if the lock count is decreased to zero. Other functions also can affect the lock count of a memory object. See the description of the **GlobalFlags** function for a list of the functions that affect the lock count.

Each time an application calls **GlobalWire** for an object, it must eventually call **GlobalUnWire** for the object.

Parameter	Type/Description
hMem	**HANDLE** Identifies the segment that will be unlocked.

Return Value The return value specifies the outcome of the function. It is TRUE if the memory segment was unlocked, that is, its lock count was decreased to zero. Otherwise, it is FALSE.

GlobalWire

Syntax **LPSTR GlobalWire**(*hMem*)

This function moves a segment into low memory and locks it—a procedure that is extremely useful if an application must lock a segment for a long period of time. If a segment from the middle portion of memory is locked for a long period of time, it causes memory-management problems by reducing the size of the largest, contiguous available block of memory. The **GlobalWire** function moves a segment to the lowest possible address in memory and locks it, thereby freeing the memory area Windows uses most often.

Each time an application calls **GlobalWire** for an object, it must eventually call **GlobalUnWire** for the object. The **GlobalUnWire** function decreases the lock count for the object. Other functions also can affect the lock count of a memory object. See the description of the **GlobalFlags** function for a list of the functions that affect the lock count.

An application must not call the **GlobalUnlock** function to unlock the object.

Parameter	Type/Description
hMem	**HANDLE** Identifies the segment that will be moved and locked.

Return Value The return value points to the new segment location. It is NULL if the function failed.

GrayString

Syntax **BOOL GrayString**(*hDC*, *hBrush*, *lpOutputFunc*, *lpData*, *nCount*, *X*, *Y*, *nWidth*, *nHeight*)

This function draws gray text at the given location. The **GrayString** function draws gray text by writing the text in a memory bitmap, graying the bitmap, and then copying the bitmap to the display. The function grays the text regardless of the selected brush and background. **GrayString** uses the font currently selected for the device context specified by the *hDC* parameter.

If the *lpOutputFunc* parameter is NULL, GDI uses the **TextOut** function, and the *lpData* parameter is assumed to be a long pointer to the character string to be output. If the characters to be output cannot be handled by **TextOut** (for example, the string is stored as a bitmap), the application must supply its own output function.

Parameter	Type/Description
hDC	**HDC** Identifies the device context.
hBrush	**HBRUSH** Identifies the brush to be used for graying.
lpOutputFunc	**FARPROC** Is the procedure-instance address of the application-supplied function that will draw the string, or, if the **TextOut** function is to be used to draw the string, it is a NULL pointer. See the following "Comments" section for details.
lpData	**DWORD** Specifies a long pointer to data to be passed to the output function. If the *lpOutputFunc* parameter is NULL, *lpData* must be a long pointer to the string to be output.

Parameter	Type/Description
nCount	**int** Specifies the number of characters to be output. If the *nCount* parameter is zero, **GrayString** calculates the length of the string (assuming that *lpData* is a pointer to the string). If *nCount* is –1 and the function pointed to by *lpOutputFunc* returns zero, the image is shown but not grayed.
X	**int** Specifies the logical *x*-coordinate of the starting position of the rectangle that encloses the string.
Y	**int** Specifies the logical *y*-coordinate of the starting position of the rectangle that encloses the string.
nWidth	**int** Specifies the width (in logical units) of the rectangle that encloses the string. If the *nWidth* parameter is zero, **GrayString** calculates the width of the area, assuming *lpData* is a pointer to the string.
nHeight	**int** Specifies the height (in logical units) of the rectangle that encloses the string. If the *nHeight* parameter is zero, **GrayString** calculates the height of the area, assuming *lpData* is a pointer to the string.

Return Value

The return value specifies the outcome of the function. It is nonzero if the string is drawn. A return value of zero means that either the **TextOut** function or the application-supplied output function returned zero, or there was insufficient memory to create a memory bitmap for graying.

Comments

An application can draw grayed strings on devices that support a solid gray color, without calling the **GrayString** function. The system color COLOR_GRAYTEXT is the solid-gray system color used to draw disabled text. The application can call the **GetSysColor** function to retrieve the color value of COLOR_GRAYTEXT. If the color is other than zero (black), the application can call the **SetTextColor** to set the text color to the color value and then draw the string directly. If the retrieved color is black, the application must call **GrayString** to gray the text.

The callback function must use the Pascal calling convention and must be declared **FAR**.

Callback Function

BOOL FAR PASCAL *OutputFunc*(*hDC*, *lpData*, *nCount*)
HDC *hDC*;
DWORD *lpData*;
int *nCount*;

OutputFunc is a placeholder for the application-supplied callback function name. The actual name must be exported by including it in an **EXPORTS** statement in the application's module-definition file.

Parameter	Description
hDC	Identifies a memory device context with a bitmap of at least the width and height specified by the *nWidth* and *nHeight* parameters, respectively.
lpData	Points to the character string to be drawn.
nCount	Specifies the number of characters to be output.

Return Value

The return value must be nonzero to indicate success. Otherwise, it is zero.

Comments

This output function (*OutputFunc*) must draw an image relative to the coordinates (0,0) rather than (*X*,*Y*). The address passed as the *lpOutputFunc* parameter must be created by using the **MakeProcInstance** function, and the output function name must be exported; it must be explicitly defined in an **EXPORTS** statement of the application's module-definition file.

The MM_TEXT mapping mode must be selected before using this function.

HIBYTE

Syntax **BYTE** **HIBYTE**(*nInteger*)

This macro retrieves the high-order byte from the integer value specified by the *nInteger* parameter.

Parameter	Type/Description
nInteger	**int** Specifies the value to be converted.

Return Value The return value specifies the high-order byte of the given value.

HideCaret

Syntax **void** **HideCaret**(*hWnd*)

This function hides the caret by removing it from the display screen. Although the caret is no longer visible, it can be displayed again by using the **ShowCaret** function. Hiding the caret does not destroy its current shape.

The **HideCaret** function hides the caret only if the given window owns the caret. If the *hWnd* parameter is NULL, the function hides the caret only if a window in the current task owns the caret.

Hiding is cumulative. If **HideCaret** has been called five times in a row, **ShowCaret** must be called five times before the caret will be shown.

Parameter	Type/Description
hWnd	**HWND** Identifies the window that owns the caret, or it is NULL to indirectly specify the window in the current task that owns the caret.

Return Value None.

HiliteMenuItem

Syntax **BOOL HiliteMenuItem**(*hWnd*, *hMenu*, *wIDHiliteItem*, *wHilite*)

This function highlights or removes the highlighting from a top-level (menu-bar) menu item.

Parameter	Type/Description
hWnd	**HWND** Identifies the window that contains the menu.
hMenu	**HMENU** Identifies the top-level menu that contains the item to be highlighted.
wIDHiliteItem	**WORD** Specifies the integer identifier of the menu item or the offset of the menu item in the menu, depending on the value of the *wHilite* parameter.
wHilite	**WORD** Specifies whether the menu item is highlighted or the highlight is removed. It can be a combination of MF_HILITE or MF_UNHILITE with MF_BYCOMMAND or MF_BYPOSITION. The values can be combined using the bitwise OR operator. These values have the following meanings:

Value	Meaning
MF_BYCOMMAND	Interprets *wIDHiliteItem* as the menu-item ID (the default interpretation).
MF_BYPOSITION	Interprets *wIDHiliteItem* as an offset.
MF_HILITE	Highlights the item. If this value is not given, highlighting is removed from the item.
MF_UNHILITE	Removes highlighting from the item.

Return Value The return value specifies whether or not the menu item is highlighted the outcome of the function. It is nonzero if the item is highlightedwas set to the specified highlight state. Otherwise, it is zero FALSE.

Comments The MF_HILITE and MF_UNHILITE flags can be used only with the **HiliteMenuItem** function; they cannot be used with the **ModifyMenu** function.

HIWORD

Syntax **WORD** **HIWORD**(*dwInteger*)

This macro retrieves the high-order word from the 32-bit integer value specified by the *dwInteger* parameter.

Parameter	Type/Description
dwInteger	**DWORD** Specifies the value to be converted.

Return Value The return value specifies the high-order word of the given 32-bit integer value.

InflateRect

Syntax

void InflateRect(*lpRect*, *X*, *Y*)

This function increases or decreases the width and height of the specified rectangle. The **InflateRect** function adds *X* units to the left and right ends of the rectangle, and adds *Y* units to the top and bottom. The *X* and *Y* parameters are signed values; positive values increase the width and height, and negative values decrease them.

Parameter	Type/Description
lpRect	**LPRECT** Points to the **RECT** data structure to be modified.
X	**int** Specifies the amount to increase or decrease the rectangle width. It must be negative to decrease the width.
Y	**int** Specifies the amount to increase or decrease the rectangle height. It must be negative to decrease the height.

Return Value

None.

Comments

The coordinate values of a rectangle must not be greater than 32,767 units or less than –32,768 units. The *X* and *Y* parameters must be chosen carefully to prevent invalid rectangles.

InitAtomTable

Syntax

BOOL InitAtomTable(*nSize*)

This function initializes an atom hash table and sets its size to that specified by the *nSize* parameter. If this function is not called, the atom hash table size is set to 37 by default.

If used, this function should be called before any other atom-management function.

Parameter	Type/Description
nSize	**int** Specifies the size (in table entries) of the atom hash table. This value should be a prime number.

Return Value

The return value specifies the outcome of the function. It is nonzero if the function is successful. Otherwise, it is zero.

Comments

If an application uses a large number of atoms, it can reduce the time required to add an atom to the atom table or to find an atom in the table by increasing the size of the table. However, this increases the amount of memory required to maintain the table.

The size of the global atom table cannot be changed from its default size of 37.

InSendMessage

Syntax

BOOL InSendMessage( )

This function specifies whether the current window function is processing a message that is passed to it through a call to the **SendMessage** function.

This function has no parameters.

Return Value

The return value specifies the outcome of the function. It is TRUE if the window function is processing a message sent to it with **SendMessage**. Otherwise, it is FALSE.

Comments

Applications use the **InSendMessage** function to determine how to handle errors that occur when an inactive window processes messages. For example, if the active window uses **SendMessage** to send a request for information to another window, the other window cannot become active until it returns control from the **SendMessage** call. The only method an inactive window has to inform the user of an error is to create a message box.

InsertMenu 3.0

Syntax

BOOL InsertMenu(*hMenu*, *nPosition*, *wFlags*, *wIDNewItem*, *lpNewItem*)

This function inserts a new menu item at the position specified by the *nPosition* parameter, moving other items down the menu. The application can specify the state of the menu item by setting values in the *wFlags* parameter.

Parameter	Type/Description
hMenu	**HMENU** Identifies the menu to be changed.
nPosition	**WORD** Specifies the menu item before which the new menu item is to be inserted. The interpretation of the *nPosition* parameter depends upon the setting of the *wFlags* parameter.

Parameter	Type/Description	
	If *wFlags* **is:**	*nPosition:*
	MF_BYPOSITION	Specifies the position of the existing menu item. The first item in the menu is at position zero. If *nPosition* is –1, the new menu item is appended to the end of the menu.
	MF_BYCOMMAND	Specifies the command ID of the existing menu item.
wFlags	**WORD** Specifies how the *nPosition* parameter is interpreted and information about the state of the new menu item when it is added to the menu. It consists of one or more values listed in the following "Comments" section.	
wIDNewItem	**WORD** Specifies either the command ID of the new menu item or, if *wFlags* is set to MF_POPUP, the menu handle of the pop-up menu.	
lpNewItem	**LPSTR** Specifies the content of the new menu item. If *wFlags* is set to MF_STRING (the default), then *lpNewItem* is a long pointer to a null-terminated character string. If *wFlags* is set to MF_BITMAP instead, then *lpNewItem* contains a bitmap handle (**HBITMAP**) in its low-order word. If *wFlags* is set to MF_OWNERDRAW, *lpNewItem* specifies an application-supplied 32-bit value which the application can use to maintain additional data associated with the menu item. This 32-bit value is available to the application in the **itemData** field of the data structure pointed to by the *lParam* parameter of the following messages: WM_MEASUREITEM WM_DRAWITEM These messages are sent when the menu item is initially displayed, or is changed.	

Return Value

The return value specifies the outcome of the function. It is TRUE if the function is successful. Otherwise, it is FALSE.

Comments

Whenever a menu changes (whether or not the menu resides in a window that is displayed), the application should call **DrawMenuBar**.

Each of the following groups lists flags that should not be used together:

- MF_BYCOMMAND and MF_BYPOSITION
- MF_DISABLED, MF_ENABLED, and MF_GRAYED
- MF_BITMAP, MF_STRING, MF_OWNERDRAW, and MF_SEPARATOR
- MF_MENUBARBREAK and MF_MENUBREAK
- MF_CHECKED and MF_UNCHECKED

The following list describes the flags which may be set in the *wFlags* parameter:

Value	Meaning
MF_BITMAP	Uses a bitmap as the item. The low-order word of the *lpNewItem* parameter contains the handle of the bitmap.
MF_BYCOMMAND	Specifies that the *nPosition* parameter gives the menu-item control ID number (default).
MF_BYPOSITION	Specifies that the *nPosition* parameter gives the position of the menu item to be changed rather than an ID number.
MF_CHECKED	Places a checkmark next to the menu item. If the application has supplied checkmark bitmaps (see the **SetMenuItemBitmaps** function), setting this flag displays the "checkmark on" bitmap next to the menu item.
MF_DISABLED	Disables the menu item so that it cannot be selected, but does not gray it.
MF_ENABLED	Enables the menu item so that it can be selected and restores it from its grayed state.
MF_GRAYED	Disables the menu item so that it cannot be selected and grays it.
MF_MENUBARBREAK	Same as MF_MENUBREAK except that for pop-up menus, separates the new column from the old column with a vertical line.
MF_MENUBREAK	Places the menu item on a new line for static menu-bar items. For pop-up menus, places the menu item in a new column, with no dividing line between the columns.

Value	Meaning
MF_OWNERDRAW	Specifies that the item is an owner-draw item. The window that owns the menu receives a WM_MEASUREITEM message when the menu is displayed for the first time to retrieve the height and width of the menu item. The WM_DRAWITEM message is then sent to the owner whenever the owner must update the visual appearance of the menu item. This option is not valid for a top-level menu item.
MF_POPUP	Specifies that the menu item has a pop-up menu associated with it. The *wIDNewItem* parameter specifies a handle to a pop-up menu to be associated with the item. Use the MF_OWNERDRAW flag to add either a top-level pop-up menu or a hierarchical pop-up menu to a pop-up menu item.
MF_SEPARATOR	Draws a horizontal dividing line. You can use this flag in a pop-up menu. This line cannot be grayed, disabled, or highlighted. Windows ignores the *lpNewItem* and *wIDNewItem* parameters.
MF_STRING	Specifies that the menu item is a character string; the *lpNewItem* parameter points to the string for the item.
MF_UNCHECKED	Does not place a checkmark next to the item (default). If the application has supplied checkmark bitmaps (see **SetMenuItemBitmaps**), setting this flag displays the "checkmark off" bitmap next to the menu item.

IntersectClipRect

Syntax **int IntersectClipRect**(*hDC, X1, Y1, X2, Y2*)

This function creates a new clipping region by forming the intersection of the current region and the rectangle specified by *X1*, *Y1*, *X2*, and *Y2*. GDI clips all subsequent output to fit within the new boundary.

Parameter	Type/Description
hDC	**HDC** Identifies the device context.
X1	**int** Specifies the logical *x*-coordinate of the upper-left corner of the rectangle.

Parameter	Type/Description
Y1	**int** Specifies the logical *y*-coordinate of the upper-left corner of the rectangle.
X2	**int** Specifies the logical *x*-coordinate of the lower-right corner of the rectangle.
Y2	**int** Specifies the logical *y*-coordinate of the lower-right corner of the rectangle.

Return Value The return value specifies the new clipping region's type. It can be any one of the following values:

Value	Meaning
COMPLEXREGION	New clipping region has overlapping borders.
ERROR	Device context is not valid.
NULLREGION	New clipping region is empty.
SIMPLEREGION	New clipping region has no overlapping borders.

Comments The width of the rectangle, specified by the absolute value of $X2 - X1$, must not exceed 32,767 units. This limit applies to the height of the rectangle as well.

H–L

IntersectRect

Syntax **int IntersectRect**(*lpDestRect*, *lpSrc1Rect*, *lpSrc2Rect*)

This function creates the intersection of two existing rectangles. The intersection is the largest rectangle contained in both rectangles. The **IntersectRect** function copies the new rectangle to the **RECT** data structure pointed to by the *lpDestRect* parameter.

Parameter	Type/Description
lpDestRect	**LPRECT** Points to the **RECT** data structure that is to receive the intersection.
lpSrc1Rect	**LPRECT** Points to a **RECT** data structure that contains a source rectangle.
lpSrc2Rect	**LPRECT** Points to a **RECT** data structure that contains a source rectangle.

Return Value The return value specifies the intersection of two rectangles. It is nonzero if the intersection of the two rectangles is not empty. It is zero if the intersection is empty.

InvalidateRect

Syntax **void InvalidateRect**(*hWnd*, *lpRect*, *bErase*)

This function invalidates the client area within the given rectangle by adding that rectangle to the window's update region. The invalidated rectangle, along with all other areas in the update region, is marked for painting when the next WM_PAINT message occurs. The invalidated areas accumulate in the update region until the region is processed when the next WM_PAINT message occurs, or the region is validated by using the **ValidateRect** or **ValidateRgn** function.

The *bErase* parameter specifies whether the background within the update area is to be erased when the update region is processed. If *bErase* is nonzero, the background is erased when the **BeginPaint** function is called; if *bErase* is zero, the background remains unchanged. If *bErase* is nonzero for any part of the update region, the background in the entire region is erased, not just in the given part.

Parameter	Type/Description
hWnd	**HWND** Identifies the window whose update region is to be modified.
lpRect	**LPRECT** Points to a **RECT** data structure that contains the rectangle (in client coordinates) to be added to the update region. If the *lpRect* parameter is NULL, the entire client area is added to the region.
bErase	**BOOL** Specifies whether the background within the update region is to be erased.

Return Value None.

Comments Windows sends a WM_PAINT message to a window whenever its update region is not empty and there are no other messages in the application queue for that window.

InvalidateRgn

Syntax **void InvalidateRgn**(*hWnd*, *hRgn*, *bErase*)

This function invalidates the client area within the given region by adding it to the current update region of the given window. The invalidated region, along with all other areas in the update region, is marked for painting when the next WM_PAINT message occurs. The invalidated areas accumulate in the update region until the region is processed when the next WM_PAINT message occurs, or the region is validated by using the **ValidateRect** or **ValidateRgn** function.

The *bErase* parameter specifies whether the background within the update area is to be erased when the update region is processed. If *bErase* is nonzero, the background is erased when the **BeginPaint** function is called; if *bErase* is zero, the background remains unchanged. If *bErase* is nonzero for any part of the update region, the background in the entire region is erased, not just in the given part.

Parameter	Type/Description
hWnd	**HWND** Identifies the window whose update region is to be modified.
hRgn	**HRGN** Identifies the region to be added to the update region. The region is assumed to have client coordinates.
bErase	**BOOL** Specifies whether the background within the update region is to be erased.

Return Value

None.

Comments

Windows sends a WM_PAINT message to a window whenever its update region is not empty and there are no other messages in the application queue for that window.

The given region must have been previously created by using one of the region functions (for more information, see Chapter 1, "Window Manager Interface Functions").

InvertRect

Syntax

void InvertRect(*hDC*, *lpRect*)

This function inverts the contents of the given rectangle. On monochrome displays, the **InvertRect** function makes white pixels black, and black pixels white. On color displays, the inversion depends on how colors are generated for the display. Calling **InvertRect** twice with the same rectangle restores the display to its previous colors.

Parameter	Type/Description
hDC	**HDC** Identifies the device context.
lpRect	**LPRECT** Points to a **RECT** data structure that contains the logical coordinates of the rectangle to be inverted.

Return Value None.

Comments The **InvertRect** function compares the values of the **top**, **bottom**, **left**, and **right** fields of the specified rectangle. If **bottom** is less than or equal to **top**, or if **right** is less than or equal to **left**, the rectangle is not drawn.

InvertRgn

Syntax **BOOL InvertRgn**(*hDC*, *hRgn*)

This function inverts the colors in the region specified by the *hRgn* parameter. On monochrome displays, the **InvertRgn** function makes white pixels black, and black pixels white. On color displays, the inversion depends on how the colors are generated for the display.

Parameter	Type/Description
hDC	**HDC** Identifies the device context for the region.
hRgn	**HRGN** Identifies the region to be filled. The coordinates for the region are specified in device units.

Return Value The return value specifies the outcome of the function. It is nonzero if the function is successful. Otherwise, it is zero.

IsCharAlpha 3.0

Syntax **BOOL IsCharAlpha**(*cChar*)

This function determines whether a character is an alphabetical character. This determination is made by the language driver based on the criteria of the current language selected by the user at setup or with the Control Panel.

Parameter	Type/Description
cChar	**char** Specifies the character to be tested.

Return Value The return value is TRUE if the character is alphabetical. Otherwise, it is FALSE.

IsCharAlphaNumeric 3.0

Syntax **BOOL IsCharAlphaNumeric**(*cChar*)

This function determines whether a character is an alphabetical or numerical character. This determination is made by the language driver based on the criteria of the current language selected by the user at setup or with the Control Panel.

Parameter	Type/Description
cChar	**char** Specifies the character to be tested.

Return Value The return value is TRUE if the character is an alphanumeric character. Otherwise, it is FALSE.

IsCharLower 3.0

Syntax **BOOL IsCharLower**(*cChar*)

This function determines whether a character is a lowercase character. This determination is made by the language driver based on the criteria of the current language selected by the user at setup or with the Control Panel.

Parameter	Type/Description
cChar	**char** Specifies the character to be tested.

Return Value The return value is TRUE if the character is lowercase. Otherwise, it is FALSE.

IsCharUpper 3.0

Syntax **BOOL IsCharUpper**(*cChar*)

This function determines whether a character is an uppercase character. This determination is made by the language driver based on the criteria of the current language selected by the user at setup or with the Control Panel.

Parameter	Type/Description
cChar	**char** Specifies the character to be tested.

Return Value The return value is TRUE if the character is uppercase. Otherwise, it is FALSE.

IsChild

Syntax **BOOL IsChild**(*hWndParent*, *hWnd*)

This function indicates whether the window specified by the *hWnd* parameter is a child window or other direct descendant of the window specified by the *hWndParent* parameter. A child window is the direct descendant of a given parent window if that parent window is in the chain of parent windows that leads from the original pop-up window to the child window.

Parameter	Type/Description
hWndParent	**HWND** Identifies a window.
hWnd	**HWND** Identifies the window to be checked.

Return Value The return value specifies the outcome of the function. It is TRUE if the window identified by the *hWnd* parameter is a child window of the window identified by the *hWndParent* parameter. Otherwise, it is FALSE.

IsClipboardFormatAvailable

Syntax **BOOL IsClipboardFormatAvailable**(*wFormat*)

This function specifies whether data of a certain type exist in the clipboard.

Parameter	Type/Description
wFormat	**WORD** Specifies a registered clipboard format. For information on clipboard formats, see the description of the **SetClipboardData** function, later in this chapter.

Return Value The return value specifies the outcome of the function. It is TRUE if data having the specified format are present. Otherwise, it is FALSE.

Comments This function is typically called during processing of the WM_INITMENU or WM_INITMENUPOPUP message to determine whether the clipboard contains data that the application can paste. If such data are present, the application typically enables the Paste command (in its Edit menu).

IsDialogMessage

Syntax **BOOL IsDialogMessage**(*hDlg*, *lpMsg*)

This function determines whether the given message is intended for the modeless dialog box specified by the *hDlg* parameter, and automatically processes the message if it is. When the **IsDialogMessage** function processes a message, it checks for keyboard messages and converts them into selection commands for the corresponding dialog box. For example, the TAB key selects the next control or group of controls, and the DOWN key selects the next control in a group.

If a message is processed by **IsDialogMessage**, it must not be passed to the **TranslateMessage** or **DispatchMessage** function. This is because **IsDialogMessage** performs all necessary translating and dispatching of messages.

IsDialogMessage sends WM_GETDLGCODE messages to the dialog function to determine which keys should be processed.

Parameter	Type/Description
hDlg	**HWND** Identifies the dialog box.
lpMsg	**LPMSG** Points to an **MSG** data structure that contains the message to be checked.

Return Value The return value specifies whether or not the given message has been processed. It is nonzero if the message has been processed. Otherwise, it is zero.

Comments Although **IsDialogMessage** is intended for modeless dialog boxes, it can be used with any window that contains controls to provide the same keyboard selection as in a dialog box.

IsDlgButtonChecked

Syntax **WORD IsDlgButtonChecked**(*hDlg*, *nIDButton*)

This function determines whether a button control has a checkmark next to it, and whether a three-state button control is grayed, checked, or neither. The **IsDlgButtonChecked** function sends a BM_GETCHECK message to the button control.

Parameter	Type/Description
hDlg	**HWND** Identifies the dialog box that contains the button control.
nIDButton	**int** Specifies the integer identifier of the button control.

Return Value The return value specifies the outcome of the function. It is nonzero if the given control has a checkmark next to it. Otherwise, it is zero. For three-state buttons, the return value is 2 if the button is grayed, 1 if the button has a checkmark next to it, and zero otherwise.

IsIconic

Syntax BOOL **IsIconic**(*hWnd*)

This function specifies whether a window is minimized (iconic).

Parameter	Type/Description
hWnd	**HWND** Identifies the window.

Return Value The return value specifies whether the window is minimized. It is nonzero if the window is minimized. Otherwise, it is zero.

IsRectEmpty

Syntax BOOL **IsRectEmpty**(*lpRect*)

This function determines whether or not the specified rectangle is empty. A rectangle is empty if the width and/or height are zero.

Parameter	Type/Description
lpRect	**LPRECT** Points to a **RECT** data structure that contains the specified rectangle.

Return Value The return value specifies whether or not the given rectangle is empty. It is nonzero if the rectangle is empty. It is zero if the rectangle is not empty.

IsWindow

Syntax

BOOL IsWindow(*hWnd*)

This function determines whether the window identified by the *hWnd* parameter is a valid, existing window.

Parameter	Type/Description
hWnd	**HWND** Identifies the window.

Return Value

The return value specifies whether or not the given window is valid. It is nonzero if *hWnd* is a valid window. Otherwise, it is zero.

IsWindowEnabled

Syntax

BOOL IsWindowEnabled(*hWnd*)

This function specifies whether the specified window is enabled for mouse and keyboard input.

Parameter	Type/Description
hWnd	**HWND** Identifies the window.

Return Value

The return value specifies whether or not the given window is enabled. It is nonzero if the window is enabled. Otherwise, it is zero.

Comments

A child window receives input only if it is both enabled and visible.

IsWindowVisible

Syntax **BOOL IsWindowVisible**(*hWnd*)

The **IsWindowVisible** function returns nonzero anytime an application has made a window visible by using the **ShowWindow** function (even if the specified window is completely covered by another child or pop-up window, the return value is nonzero).

Parameter	Type/Description
hWnd	**HWND** Identifies the window.

Return Value The return value specifies whether or not a given window exists on the screen. It is nonzero if the given window exists on the screen. Otherwise, it is zero.

IsZoomed

Syntax **BOOL IsZoomed**(*hWnd*)

This function determines whether or not a window has been maximized.

Parameter	Type/Description
hWnd	**HWND** Identifies the window.

Return Value The return value specifies whether or not the given window is maximized. It is nonzero if the window is maximized. Otherwise, it is zero.

KillTimer

Syntax

BOOL KillTimer(*hWnd*, *nIDEvent*)

This function kills the timer event identified by the *hWnd* and *nIDEvent* parameters. Any pending WM_TIMER messages associated with the timer are removed from the message queue.

Parameter	Type/Description
hWnd	**HWND** Identifies the window associated with the given timer event. This must be the same value passed as the *hWnd* parameter to the **SetTimer** function call that created the timer event.
nIDEvent	**int** Specifies the timer event to be killed. If the application called **SetTimer** with the *hWnd* parameter set to NULL, this must be the event identifier returned by **SetTimer**. If the *hWnd* parameter of **SetTimer** was a valid window handle, *nIDEvent* must be the value of the *nIDEvent* parameter passed to **SetTimer**.

Return Value

The return value specifies the outcome of the function. It is nonzero if the event was killed. It is zero if the **KillTimer** function could not find the specified timer event.

_lclose

Syntax

int _lclose(*hFile*)

This function closes the file specified by the *hFile* parameter. As a result, the file is no longer available for reading or writing.

The *hFile* argument is returned by the call that created or last opened the file.

Parameter	Type/Description
hFile	**int** Specifies the MS-DOS file handle of the file to be closed.

Return Value

The return value indicates whether the function successfully closed the file. It is zero if the function closed the file, or –1 if the function failed.

_lcreat

Syntax

int _lcreat(*lpPathName*, *iAttribute*)

This function opens a file with the name specified by the *lpPathName* parameter. The *iAttribute* parameter specifies the attributes of the file when the function opens it. If the file does not exist, the function creates a new file and opens it for writing. If the file does exist, the function truncates the file size to zero and opens it for reading and writing. When the function opens the file, the pointer is set to the beginning of the file.

Parameter	Type/Description
lpPathName	**LPSTR** Points to a null-terminated character string that names the file to be opened. The string must consist of characters from the ANSI character set.
iAttribute	**int** Specifies the file attributes. The parameter must be one of these values:

Value	Meaning
0	Normal; can be read or written without restriction.
1	Read-only; cannot be opened for write; a file with the same name cannot be created.
2	Hidden; not found by directory search.
3	System; not found by directory search.

Return Value The return value specifies an MS-DOS file handle if the function was successful. Otherwise, the return value is –1.

LimitEmsPages

Syntax **void LimitEmsPages** (*dwKbytes*)

This function limits the amount of expanded memory that Windows will assign to an application. It does not limit the amount of expanded memory that the application can get by directly calling INT 67H.

Parameter	Type/Description
dwKbytes	**DWORD** Specifies the number of kilobytes of expanded memory to which the application is to have access.

Return value None.

Comments **LimitEmsPages** has an effect only if expanded memory is installed and being used by Windows. If Windows is not using expanded memory, then the function has no effect.

LineDDA

Syntax **void LineDDA**(*X1, Y1, X2, Y2, lpLineFunc, lpData*)

This function computes all successive points in a line starting at the point specified by the *X1* and *Y1* parameters and ending at the point specified by the *X2* and *Y2* parameters. The endpoint is not included as part of the line. For each point on the line, the **LineDDA** function calls the application-supplied function pointed to by the *lpLineFunc* parameter, passing to the function the coordinates of the current point and the *lpData* parameter.

Parameter	Type/Description
X1	**int** Specifies the logical *x*-coordinate of the first point.
Y1	**int** Specifies the logical *y*-coordinate of the first point.
X2	**int** Specifies the logical *x*-coordinate of the endpoint.
Y2	**int** Specifies the logical *y*-coordinate of the endpoint.
lpLineFunc	**FARPROC** Is the procedure-instance address of the application-supplied function. See the following "Comments" section for details.

Parameter	Type/Description
lpData	**LPSTR** Points to the application-supplied data.

Return Value None.

Comments The address passed by the *lpLineFunc* parameter must be created by using the **MakeProcInstance** function.

The callback function must use the Pascal calling convention and must be declared **FAR**.

Callback Function

```
void FAR PASCAL LineFunc(X, Y, lpData)
int X;
int Y;
LPSTR lpData;
```

LineFunc is a placeholder for the application-supplied function name. The actual name must be exported by including it in an **EXPORTS** statement in the application's module-definition file.

Parameter	Definition
X	Specifies the *x*-coordinate of the current point.
Y	Specifies the *y*-coordinate of the current point.
lpData	Points to the application-supplied data.

Return Value

The function can perform any task. It has no return value.

LineTo

Syntax **BOOL LineTo**(*hDC*, *X*, *Y*)

This function draws a line from the current position up to, but not including, the point specified by the *X* and *Y* parameters. The line is drawn with the selected pen. If no error occurs, the position is set to (*X*,*Y*).

Parameter	Type/Description
hDC	**HDC** Identifies the device context.

Parameter	Type/Description
X	**int** Specifies the logical *x*-coordinate of the endpoint for the line.
Y	**int** Specifies the logical *y*-coordinate of the endpoint for the line.

Return Value The return value specifies whether or not the line is drawn. It is nonzero if the line is drawn. Otherwise, it is zero.

_llseek

Syntax LONG **_llseek**(*hFile*, *lOffset*, *iOrigin*)

This function repositions the pointer in a previously opened file. The *iOrigin* parameter specifies the starting position in the file, and *lOffset* specifies how far (in bytes) the function is to move the pointer.

Parameter	Type/Description
hFile	**int** Specifies the MS-DOS file handle of the file.
lOffset	**LONG** Specifies the number of bytes the pointer is to be moved.
iOrigin	**int** Specifies the starting position and direction of the pointer. The parameter must be one of the following values:

Value	Meaning
0	Move the file pointer *lOffset* bytes from the beginning of the file.
1	Move the file pointer *lOffset* bytes from the current position of the file.
2	Move the file pointer *lOffset* bytes from the end of the file.

Return Value The return value specifies the new offset of the pointer (in bytes) from the beginning of the file. The return value is –1 if the function fails.

Comments When a file is initially opened, the file pointer is positioned at the beginning of the file. The **_llseek** function permits random access to a file's contents by moving the pointer an arbitrary amount without reading data.

LoadAccelerators

Syntax

HANDLE **LoadAccelerators**(*hInstance*, *lpTableName*)

This function loads the accelerator table named by the *lpTableName* parameter from the executable file associated with the module specified by the *hInstance* parameter.

The **LoadAccelerators** function loads the table only if it has not been previously loaded. Otherwise, it retrieves a handle to the loaded table.

Parameter	Type/Description
hInstance	**HANDLE** Identifies an instance of the module whose executable file contains the accelerator table.
lpTableName	**LPSTR** Points to a string that names the accelerator table. The string must be a null-terminated character string.

Return Value

The return value identifies the loaded accelerator table if the function is successful. Otherwise, it is NULL.

LoadBitmap

Syntax

HBITMAP **LoadBitmap**(*hInstance*, *lpBitmapName*)

This function loads the bitmap resource named by the *lpBitmapName* parameter from the executable file associated with the module specified by the *hInstance* parameter.

Parameter	Type/Description
hInstance	**HANDLE** Identifies the instance of the module whose executable file contains the bitmap.
lpBitmapName	**LPSTR** Points to a character string that names the bitmap. The string must be a null-terminated character string.

Return Value

The return value identifies the specified bitmap. It is NULL if no such bitmap exists.

Comments

The application must call the **DeleteObject** function to delete each bitmap handle returned by the **LoadBitmap** function. This also applies to the predefined bitmaps described in the following paragraph.

The **LoadBitmap** function can also be used to access the predefined bitmaps used by Windows. The *hInstance* parameter must be set to NULL, and the *lpBitmapName* parameter must be one of the following values:

- OBM_BTNCORNERS
- OBM_BTSIZE
- OBM_CHECK
- OBM_CHECKBOXES
- OBM_CLOSE
- OBM_COMBO
- OBM_DNARROW
- OBM_DNARROWD
- OBM_LFARROW
- OBM_LFARROWD
- OBM_MNARROW
- OBM_OLD_CLOSE
- OBM_OLD_DNARROW
- OBM_OLD_LFARROW
- OBM_OLD_REDUCE
- OBM_OLD_RESTORE
- OBM_OLD_RGARROW
- OBM_OLD_UPARROW
- OBM_OLD_ZOOM
- OBM_REDUCE
- OBM_REDUCED
- OBM_RESTORE
- OBM_RESTORED
- OBM_RGARROW
- OBM_RGARROWD

- OBM_SIZE
- OBM_UPARROW
- OBM_UPARROWD
- OBM_ZOOM
- OBM_ZOOMD

Bitmap names that begin OBM_OLD represent bitmaps used by Windows versions prior to 3.0.

The *lpBitmapName* parameter can also be a value created by the **MAKEINTRESOURCE** macro. If it is, the ID must reside in the low-order word of *lpBitmapName*, and the high-order word must contain zeros.

LoadCursor

Syntax

HCURSOR LoadCursor(*hInstance*, *lpCursorName*)

This function loads the cursor resource named by the *lpCursorName* parameter from the executable file associated with the module specified by the *hInstance* parameter. The function loads the cursor into memory only if it has not been previously loaded. Otherwise, it retrieves a handle to the existing resource.

Parameter	Type/Description
hInstance	**HANDLE** Identifies an instance of the module whose executable file contains the cursor.
lpCursorName	**LPSTR** Points to a character string that names the cursor resource. The string must be a null-terminated character string.

Return Value

The return value identifies the newly loaded cursor if the function is successful. Otherwise, it is NULL.

Comments

The **LoadCursor** function returns a valid cursor handle only if the *lpCursorName* parameter identifies a cursor resource. If *lpCursorName* identifies any type of resource other than a cursor (such as an icon), the return value will not be NULL, even though it is not a valid cursor handle.

Use the **LoadCursor** function to access the predefined cursors used by Windows. To do this, the *hInstance* parameter must be set to NULL, and the *lpCursorName* parameter must be one of the following values:

Value	Meaning
IDC_ARROW	Standard arrow cursor.
IDC_CROSS	Crosshair cursor.
IDC_IBEAM	Text I-beam cursor.
IDC_ICON	Empty icon.
IDC_SIZE	Loads a square with a smaller square inside its lower-right corner.
IDC_SIZENESW	Double-pointed cursor with arrows pointing northeast and southwest.
IDC_SIZENS	Double-pointed cursor with arrows pointing north and south.
IDC_SIZENWSE	Double-pointed cursor with arrows pointing northwest and southeast.
IDC_SIZEWE	Double-pointed cursor with arrows pointing west and east.
IDC_UPARROW	Vertical arrow cursor.
IDC_WAIT	Hourglass cursor.

The *lpCursorName* parameter can contain a value created by the **MAKEINTRESOURCE** macro. If it does, the ID must reside in the low-order word of *lpCursorName*, and the high-order word must be set to zero.

LoadIcon

Syntax **HICON LoadIcon**(*hInstance*, *lpIconName*)

This function loads the icon resource named by the *lpIconName* parameter from the executable file associated with the module specified by the *hInstance* parameter. The function loads the icon only if it has not been previously loaded. Otherwise, it retrieves a handle to the loaded resource.

Parameter	Type/Description
hInstance	**HANDLE** Identifies an instance of the module whose executable file contains the icon.
lpIconName	**LPSTR** Points to a character string that names the icon resource. The string must be a null-terminated character string.

Return Value The return value identifies an icon resource if the function is successful. Otherwise, it is NULL.

Comments Use the **LoadIcon** function to access the predefined icons used by Windows. To do this, the *hInstance* parameter must be set to NULL, and the *lpIconName* parameter must be one of the following values:

Value	Meaning
IDI_APPLICATION	Default application icon.
IDI_ASTERISK	Asterisk (used in informative messages).
IDI_EXCLAMATION	Exclamation point (used in warning messages).
IDI_HAND	Hand-shaped icon (used in serious warning messages).
IDI_QUESTION	Question mark (used in prompting messages).

The *lpIconName* parameter can also contain a value created by the **MAKEINTRESOURCE** macro. If it does, the ID must reside in the low-order word of *lpIconName*, and the high-order word must be set to zero.

LoadLibrary

Syntax **HANDLE LoadLibrary**(*lpLibFileName*)

This function loads the library module contained in the specified file and retrieves a handle to the loaded module instance.

Parameter	Type/Description
lpLibFileName	**LPSTR** Points to a string that names the library file. The string must be a null-terminated character string.

Return Value The return value identifies the instance of the loaded library module. Otherwise, it is a value less than 32 that specifies the error. The following list describes the error values returned by this function:

Value	Meaning
0	Out of memory.
2	File not found.

Value	Meaning
3	Path not found.
5	Attempt to dynamically link to a task.
6	Library requires separate data segments for each task.
10	Incorrect Windows version.
11	Invalid .EXE file (non-Windows .EXE or error in .EXE image).
12	OS/2 application.
13	DOS 4.0 application.
14	Unknown .EXE type.
15	Attempt in protected (standard or 386 enhanced) mode to load an .EXE created for an earlier version of Windows.
16	Attempt to load a second instance of an .EXE containing multiple, writeable data segments.
17	Attempt in large-frame EMS mode to load a second instance of an application that links to certain nonshareable DLLs already in use.
18	Attempt in real mode to load an application marked for protected mode only.

LoadMenu

Syntax **HMENU LoadMenu**(*hInstance*, *lpMenuName*)

This function loads the menu resource named by the *lpMenuName* parameter from the executable file associated with the module specified by the *hInstance* parameter.

Parameter	Type/Description
hInstance	**HANDLE** Identifies an instance of the module whose executable file contains the menu.
lpMenuName	**LPSTR** Points to a character string that names the menu resource. The string must be a null-terminated character string.

Return Value The return value identifies a menu resource if the function is successful. Otherwise, it is NULL.

Comments The *lpMenuName* parameter can contain a value created by the **MAKEINTRESOURCE** macro. If it does, the ID must reside in the low-order word of *lpMenuName*, and the high-order word must be set to zero.

LoadMenuIndirect

Syntax HMENU **LoadMenuIndirect**(*lpMenuTemplate*)

This function loads into memory the menu named by the *lpMenuTemplate* parameter. The template specified by *lpMenuTemplate* is a header followed by a collection of one or more **MENUITEMTEMPLATE** structures, each of which may contain one or more menu items and pop-up menus.

Parameter	Type/Description
lpMenuTemplate	**LPSTR** Points to a menu template (which is a collection of one or more **MENUITEMTEMPLATE** structures).

Return Value The return value identifies the menu if the function is successful. Otherwise, it is NULL.

LoadModule 3.0

Syntax HANDLE **LoadModule**(*lpModuleName*, *lpParameterBlock*)

This function loads and executes a Windows program or creates a new instance of an existing Windows program.

Parameter	Type/Description
lpModuleName	**LPSTR** Points to a null-terminated string that contains the filename of the application to be run. If the *lpModuleName* string does not contain a directory path, Windows will search for the executable file in this order:

1. The current directory
2. The Windows directory (the directory containing WIN.COM); the **GetWindowsDirectory** function obtains the pathname of this directory

Parameter	Type/Description
	3. The Windows system directory (the directory containing such system files as KERNEL.EXE); the **GetSystem-Directory** function obtains the pathname of this directory 4. The directories listed in the PATH environment variable 5. The list of directories mapped in a network If the application filename does not contain an extension, then .EXE is assumed.
lpParameterBlock	**LPVOID** Points to a data structure consisting of four fields that defines a parameter block. This data structure consists of the following fields:

Field	Type/Description
wEnvSeg	**WORD** Specifies the segment address of the environment under which the module is to run; 0 indicates that the Windows environment is to be copied.
lpCmdLine	**LPSTR** Points to a null-terminated character string that contains a correctly formed command line. This string must not exceed 120 bytes in length.
lpCmdShow	**LPVOID** Points to a data structure containing two **WORD**-length values. The first value must always be set to two. The second value specifies how the application window is to be shown. See the description of the *nCmdShow* paramter of the **ShowWindow** function for a list of the acceptable values.
dwReserved	**DWORD** Is reserved and must be NULL.

All unused fields should be set to NULL, except for *lpCmdLine*, which must point to a null string if it is not used.

Return Value

The return value identifies the instance of the loaded module if the function was successful. Otherwise, it is a value less than 32 that specifies the error. The following list describes the error values returned by this function:

Value	Meaning
0	Out of memory.
2	File not found.
3	Path not found.
5	Attempt to dynamically link to a task.
6	Library requires separate data segments for each task.
10	Incorrect Windows version.
11	Invalid .EXE file (non-Windows .EXE or error in .EXE image).
12	OS/2 application.
13	DOS 4.0 application.
14	Unknown .EXE type.
15	Attempt in protected (standard or 386 enhanced) mode to load an .EXE created for an earlier version of Windows.
16	Attempt to load a second instance of an .EXE containing multiple, writeable data segments.
17	Attempt in large-frame EMS mode to load a second instance of an application that links to certain nonshareable DLLs already in use.
18	Attempt in real mode to load an application marked for protected mode only.

Comments

The **WinExec** function provides an alternative method for executing a program.

LoadResource

Syntax

HANDLE LoadResource(*hInstance*, *hResInfo*)

This function loads a resource identified by the *hResInfo* parameter from the executable file associated with the module specified by the *hInstance* parameter. The function loads the resource into memory only if it has not been previously loaded. Otherwise, it retrieves a handle to the existing resource.

Parameter	Type/Description
hInstance	**HANDLE** Identifies an instance of the module whose executable file contains the resource.
hResInfo	**HANDLE** Identifies the desired resource. This handle is assumed to have been created by using the **FindResource** function.

Return Value The return value identifies the global memory block to receive the data associated with the resource. It is NULL if no such resource exists.

Comments The resource is not actually loaded until the **LockResource** function is called to translate the handle returned by **LoadResource** into a far pointer to the resource data.

LoadString

Syntax **int LoadString**(*hInstance*, *wID*, *lpBuffer*, *nBufferMax*)

This function loads a string resource identified by the *wID* parameter from the executable file associated with the module specified by the *hInstance* parameter. The function copies the string into the buffer pointed to by the *lpBuffer* parameter, and appends a terminating null character.

Parameter	Type/Description
hInstance	**HANDLE** Identifies an instance of the module whose executable file contains the string resource.
wID	**WORD** Specifies the integer identifier of the string to be loaded.
lpBuffer	**LPSTR** Points to the buffer that receives the string.
nBufferMax	**int** Specifies the maximum number of characters to be copied to the buffer. The string is truncated if it is longer than the number of characters specified.

Return Value The return value specifies the actual number of characters copied into the buffer. It is zero if the string resource does not exist.

LOBYTE

Syntax

BYTE LOBYTE(*nInteger*)

This macro extracts the low-order byte from the short-integer value specified by the *nInteger* parameter.

Parameter	Type/Description
nInteger	**int** Specifies the value to be converted.

Return Value

The return value specifies the low-order byte of the value.

LocalAlloc

Syntax

HANDLE LocalAlloc(*wFlags*, *wBytes*)

This function allocates the number of bytes of memory specified by the *wBytes* parameter from the local heap. The memory block can be either fixed or moveable, as specified by the *wFlags* parameter.

Parameter	Type/Description
wFlags	**WORD** Specifies how to allocate memory. It can be one or more of the following values:

Value	Meaning
LMEM_DISCARDABLE	Allocates discardable memory. Can only be used with LMEM_MOVEABLE.
LMEM_FIXED	Allocates fixed memory.
LMEM_MODIFY	Modifies the LMEM_DISCARDABLE flag. Can only be used with LMEM_DISCARDABLE.
LMEM_MOVEABLE	Allocates moveable memory. Cannot be used with LMEM_FIXED.
LMEM_NOCOMPACT	Does not compact or discard memory to satisfy the allocation request.

Parameter	Type/Description	
	Value	**Meaning**
	LMEM_NODISCARD	Does not discard memory to satisfy the allocation request.
	LMEM_ZEROINIT	Initializes memory contents to zero.
	Choose LMEM_FIXED or LMEM_MOVEABLE, and then combine others as needed by using the bitwise OR operator.	
wBytes	**WORD** Specifies the total number of bytes to be allocated.	

Return Value The return value identifies the newly allocated local memory block if the function is successful. Otherwise, it is NULL.

Comments If the data segment that contains the heap is moveable, calling this function will cause the data segment to move if Windows needs to increase the size of the heap and cannot increase the size of the heap in its current location. An application can prevent Windows from moving the data segment by calling the **LockData** function to lock the data segment.

If this function is successful, it allocates at least the amount requested. The actual amount allocated may be greater. To determine the actual amount allocated, call the **LocalSize** function.

LocalCompact

Syntax **WORD LocalCompact**(*wMinFree*)

This function generates the number of free bytes of memory specified by the *wMinFree* parameter by compacting, if necessary, the module's local heap. The function checks the local heap for the specified number of contiguous free bytes. If the bytes do not exist, the **LocalCompact** function compacts local memory by first moving all unlocked moveable blocks into high memory. If this does not generate the requested amount of space, the function discards moveable and discardable blocks that are not locked down, until the requested amount of space is generated, whenever possible.

Parameter	Type/Description
wMinFree	**WORD** Specifies the number of free bytes desired. If *wMinFree* is zero, the function returns a value but does not compact memory.

Return Value The return value specifies the number of bytes in the largest block of free local memory.

LocalDiscard

Syntax **HANDLE LocalDiscard**(*hMem*)

This function discards the local memory block specified by the *hMem* parameter. The lock count of the memory block must be zero.

The local memory block is removed from memory, but its handle remains valid. An application can subsequently pass the handle to the **LocalReAlloc** function to allocate another local memory block identified by the same handle.

Parameter	Type/Description
hMem	**HANDLE** Identifies the local memory block to be discarded.

Return Value The return value specifies the outcome of the function. It is NULL if the function is successful. Otherwise, it is equal to *hMem*.

LocalFlags

Syntax **WORD LocalFlags**(*hMem*)

This function returns information about the specified local memory block.

Parameter	Type/Description
hMem	**HANDLE** Identifies the local memory block.

Return Value The return value contains one of the following memory-allocation flags in the high byte:

Value	Meaning
LMEM_DISCARDABLE	Block is marked as discardable.
LMEM_DISCARDED	Block has been discarded.

The low byte of the return value contains the reference count of the block. Use the LMEM_LOCKCOUNT mask to retrieve the lock-count value from the return value.

LocalFree

Syntax HANDLE **LocalFree**(*hMem*)

This function frees the local memory block identified by the *hMem* parameter and invalidates the handle of the memory block.

Parameter	Type/Description
hMem	**HANDLE** Identifies the local memory block to be freed.

Return Value The return value specifies the outcome of the function. It is NULL if the function is successful. Otherwise, it is equal to *hMem*.

LocalHandle

Syntax HANDLE **LocalHandle**(*wMem*)

This function retrieves the handle of the local memory object whose address is specified by the *wMem* parameter.

Parameter	Type/Description
wMem	**WORD** Specifies the address of a local memory object.

Return Value The return value identifies the local memory object.

LocalInit

Syntax BOOL **LocalInit**(*wSegment*, *pStart*, *pEnd*)

This function initializes a local heap in the segment specified by the *wSegment* parameter.

Parameter	Type/Description
wSegment	**WORD** Specifies the segment address of the segment that is to contain the local heap.
pStart	**PSTR** Specifies the address of the start of the local heap within the segment.

Parameter	Type/Description
pEnd	**PSTR** Specifies the address of the end of the local heap within the segment.

Return Value The return value specifies a Boolean value that is nonzero if the heap is initialized. Otherwise, it is zero.

Comments If the *pStart* parameter is zero, the *pEnd* parameter specifies the offset of the last byte of the global heap from the end of the segment. For example, to initialize a 4096-byte heap with the first byte at byte 0, set *pStart* to 0 and *pEnd* to 4095.

LocalInit calls the **GlobalLock** function for the data segment that contains the local heap. This ensures that the data segment will not be moved in memory. However, the memory will be moved if both of these conditions are true:

1. The data segment is moveable.
2. The application calls the **LocalAlloc** or **LocalReAlloc** function and, as a result, Windows must increase the size of the heap. If Windows cannot increase the size of the data segment that contains the local heap without moving it, Windows will move the data segment.

An application can explicitly prevent Windows from moving the data segment by calling the **LockData** function to lock the data segment.

An application can remove this initial lock count by calling the **UnlockData** function.

LocalLock

Syntax **PSTR LocalLock**(*hMem*)

This function locks the local memory block specified by the *hMem* parameter. The block is locked into memory at the given address and its reference count is increased by one. Locked memory cannot be moved or discarded. The block remains locked in memory until its reference count is decreased to zero by using the **LocalUnlock** function.

Parameter	Type/Description
hMem	**HANDLE** Identifies the local memory block to be locked.

Return Value The return value points to the first byte of memory in the local block if the function is successful. Otherwise, it is NULL.

LocalReAlloc

Syntax HANDLE **LocalReAlloc**(*hMem*, *wBytes*, *wFlags*)

This function changes the size of the local memory block specified by the *hMem* parameter by increasing or decreasing its size to the number of bytes specified by the *wBytes* parameter, or changes the attributes of the specified memory block.

Parameter	Type/Description
hMem	**HANDLE** Identifies the local memory block to be reallocated.
wBytes	**WORD** Specifies the new size of the memory block.
wFlags	**WORD** Specifies how to reallocate the local memory block. It can be one or more of the following values:

Value	Meaning
LMEM_DISCARDABLE	Memory is discardable. This flag can only be used with LMEM_MODIFY.
LMEM_MODIFY	Memory flags are modified. The *wBytes* parameter is ignored. This flag can only be used with LMEM_DISCARDABLE.
LMEM_MOVEABLE	Memory is moveable. If *wBytes* is zero, this flag causes a previously fixed block to be freed or a previously moveable object to be discarded (if the block's reference count is zero). If *wBytes* is nonzero and the block specified by *hMem* is fixed, this flag allows the reallocated block to be moved to a new fixed location. (Note that the handle returned by the **LocalReAlloc** function in this case may be different from the handle passed to the function.) This flag cannot be used with LMEM_MODIFY.
LMEM_NOCOMPACT	Memory will not be compacted or discarded to satisfy the allocation request. This flag cannot be used with LMEM_MODIFY.

Parameter	Type/Description	
	Value	**Meaning**
	LMEM_NODISCARD	Objects will not be discarded to satisfy the allocation request. This flag cannot be used with LMEM_MODIFY.
	LMEM_ZEROINIT	If the block is growing, the additional memory contents are initialized to zero. This flag cannot be used with LMEM_MODIFY.

Return Value

The return value identifies the reallocated local memory if the function is successful. It is NULL if the local memory block cannot be reallocated.

The return value is always identical to the *hMem* parameter, unless the LMEM_MOVEABLE flag is used to allow movement of a fixed block of memory to a new fixed location.

Comments

If the data segment that contains the heap is moveable, calling this function will cause the data segment to move if Windows must increase the size of the heap and cannot increase the size of the heap in its current location. An application can prevent Windows from moving the data segment by calling the **LockData** function to lock the data segment.

LocalShrink

Syntax

WORD **LocalShrink**(*hSeg*, *wSize*)

This function shrinks the specified heap to the size specified by the *wSize* parameter. The minimum size for the automatic local heap is defined in the application's module-definition file.

Parameter	Type/Description
hSeg	**HANDLE** Identifies the segment that contains the local heap.
wSize	**WORD** Specifies the size (in bytes) desired for the local heap after shrinkage.

Return Value

The return value specifies the size of the local heap after shrinkage.

Comments

If *hSeg* is zero, the **LocalShrink** function reduces the local heap in the current data segment. Windows will not shrink that portion of the data segment that contains the stack and the static variables.

Use the **GlobalSize** function to determine the new size of the data segment.

LocalSize

Syntax

WORD LocalSize(*hMem*)

This function retrieves the current size (in bytes) of the local memory block specified by the *hMem* parameter.

Parameter	Type/Description
hMem	**HANDLE** Identifies the local memory block.

Return Value

The return value specifies the size (in bytes) of the specified memory block. It is NULL if the given handle is not valid.

Comments

The actual size of a memory block sometimes is larger than the size requested when the memory was allocated.

LocalUnlock

Syntax

BOOL LocalUnlock(*hMem*)

This function unlocks the local memory block specified by the *hMem* parameter and decreases the block's reference count by one. The block is completely unlocked, and subject to moving or discarding, if the reference count is decreased to zero.

Parameter	Type/Description
hMem	**HANDLE** Identifies the local memory block to be unlocked.

Return Value

The return value is zero if the block's reference count was decreased to zero. Otherwise, the return value is nonzero.

LockData

Syntax **HANDLE LockData**(*Dummy*)

This macro locks the current data segment in memory. It is intended to be used in modules that have moveable data segments.

Parameter	Type/Description
Dummy	**int** Is not used. It should be set to zero.

Return Value The return value identifies the locked data segment if the function is successful. Otherwise, it is NULL.

LockResource

Syntax **LPSTR LockResource**(*hResData*)

This function retrieves the absolute memory address of the loaded resource identified by the *hResData* parameter. The resource is locked in memory and the given address and its reference count are increased by one. The locked resource is not subject to moving or discarding.

The resource remains locked in memory until its reference count is decreased to zero through calls to the **FreeResource** function.

If the resource identified by *hResData* has been discarded, the resource-handler function (if any) associated with the resource is called before the **LockResource** function returns. The resource-handler function can recalculate and reload the resource if desired. After the resource-handler function returns, **LockResource** makes another attempt to lock the resource and returns with the result.

Parameter	Type/Description
hResData	**HANDLE** Identifies the desired resource. This handle is assumed to have been created by using the **LoadResource** function.

Return Value The return value points to the first byte of the loaded resource if the resource was locked. Otherwise, it is NULL.

Comments Using the handle returned by the **FindResource** function for the *hResData* parameter causes an error.

Use the **UnlockResource** macro to unlock a resource that was locked by using **LockResource.**

LockSegment

Syntax **HANDLE LockSegment**(*wSegment*)

This function locks the segment whose segment address is specified by the *wSegment* parameter. If *wSegment* is –1, the **LockSegment** function locks the current data segment.

Except for nondiscardable segments in protected (standard or 386 enhanced) mode, the segment is locked into memory at the given address and its lock count is increased by one. Locked memory is not subject to moving or discarding except when a portion of the segment is being reallocated by the **GlobalReAlloc** function. The segment remains locked in memory until its lock count is decreased to zero.

In protected mode, **LockSegment** increments the lock count of discardable and automatic data segments only.

Each time an application calls **LockSegment** for a segment, it must eventually call **UnlockSegment** for the segment. The **UnlockSegment** function decreases the lock count for the segment. Other functions also can affect the lock count of a memory object. See the description of the **GlobalFlags** function for a list of the functions that affect the lock count.

Parameter	Type/Description
wSegment	**WORD** Specifies the segment address of the segment to be locked. If *wSegment* is –1, the **LockSegment** function locks the current data segment.

Return Value The return value identifies the data segment if the function is successful. If the object has been discarded or an error occurs, the return value is NULL.

_lopen

Syntax **int _lopen**(*lpPathName, iReadWrite*)

This function opens the file with the name specified by the *lpPathName* parameter. The *iReadWrite* parameter specifies the access mode of the file when the function opens it. If the file exists and is opened for writing only, the function truncates the file size to zero. When the function opens the file, the pointer is set to the beginning of the file.

Parameter	Type/Description
lpPathName	**LPSTR** Points to a null-terminated character string that names the file to be opened. The string must consist of characters from the ANSI character set.
iReadWrite	**int** Specifies whether the function is to open the file with read access, write access, or both. The parameter must be one of the following values:

Value	Meaning
OF_READ	Opens the file for reading only.
OF_READWRITE	Opens the file for reading and writing.
OF_SHARE_COMPAT	Opens the file with compatibility mode, allowing any process on a given machine to open the file any number of times. **OpenFile** fails if the file has been opened with any of the other sharing modes.
OF_SHARE_DENY_NONE	Opens the file without denying other processes read or write access to the file. **OpenFile** fails if the file has been opened in compatibility mode by any other process.
OF_SHARE_DENY_READ	Opens the file and denies other processes read access to the file. **OpenFile** fails if the file has been opened in compatibility mode or for read access by any other process.
OF_SHARE_DENY_WRITE	Opens the file and denies other processes write access to the file. **OpenFile** fails if the file has been opened in compatibility or for write access by any other process.

Parameter	Type/Description	
	Value	**Meaning**
	OF_SHARE_EXCLUSIVE	Opens the file with exclusive mode, denying other processes both read and write access to the file. **OpenFile** fails if the file has been opened in any other mode for read or write access, even by the current process.
	OF_WRITE	Opens the file for writing only.

Return Value The return value specifies an MS-DOS file handle if the function opened the file. Otherwise, it is –1.

LOWORD

Syntax **WORD LOWORD**(*dwInteger*)

This macro extracts the low-order word from the 32-bit integer value specified by the *dwInteger* parameter.

Parameter	Type/Description
dwInteger	**DWORD** Specifies the value to be converted.

Return Value The return value specifies the low-order word of the 32-bit integer value.

LPtoDP

Syntax **BOOL LPtoDP**(*hDC*, *lpPoints*, *nCount*)

This function converts logical points into device points. The **LPtoDP** function maps the coordinates of each point specified by the *lpPoints* parameter from GDI's logical coordinate system into a device coordinate system. The conversion depends on the current mapping mode.

Parameter	Type/Description
hDC	**HANDLE** Identifies the device context.
lpPoints	**LPPOINT** Points to an array of points. Each point in the array is a **POINT** data structure.
nCount	**int** Specifies the number of points in the array.

Return Value The return value specifies whether or not all points are converted. It is nonzero if all points are converted. Otherwise, it is zero.

_lread

Syntax **int _lread**(*hFile*, *lpBuffer*, *wBytes*)

This function reads data from the file identified by the *hFile* parameter. The *wBytes* parameter specifies the number of bytes to read. The function return value indicates the number of bytes actually read. The return value is zero if the function attempted to read the file at EOF.

Parameter	Type/Description
hFile	**int** Specifies the MS-DOS file handle of the file to be read.
lpBuffer	**LPSTR** Points to a buffer that is to receive the data read from the file.
wBytes	**WORD** Specifies the number of bytes to be read from the file.

Return Value The return value indicates the number of bytes which the function actually read from the file, or –1 if the function fails. The return value is less than *wBytes* if the function encountered the end of the file (EOF) before reading the specified number of bytes.

lstrcat

Syntax **LPSTR lstrcat**(*lpString1*, *lpString2*)

This function concatenates *lpString2* to the string specified by *lpString1*, terminates the resulting string with a null character, and returns a far pointer to the concatenated string (*lpString1*).

All strings must be less than 64K in size.

Parameter	Type/Description
lpString1	**LPSTR** Points to byte array containing a null-terminated string to which the function is to append *lpString2*. The byte array containing the string must be large enough to contain both strings.
lpString2	**LPSTR** Points to the null-terminated string which the function is to append to *lpString1*.

Return Value The return value specifies a pointer to *lpString1*. It is zero if the function fails.

lstrcmp 3.0

Syntax **int lstrcmp**(*lpString1*, *lpString2*)

This function compares the two strings identified by *lpString1* and *lpString2* lexicographically and returns a value indicating their relationship. The comparison is made based on the current language selected by the user at setup or with the Control Panel. The comparison is case-sensitive. This function is not equivalent to the **strcmp** C run-time library function.

All strings must be less than 64K in size.

Parameter	Type/Description
lpString1	**LPSTR** Points to the first null-terminated string to be compared.
lpString2	**LPSTR** Points to the second null-terminated string to be compared.

Return Value The return value indicates whether *lpString1* is less than, equal to, or greater than *lpString2*. This relationship is outlined in the following:

Value	Meaning
< 0	*lpString1* is less than *lpString2*.
= 0	*lpString1* is identical to *lpString2*.
> 0	*lpString1* is greater than *lpString2*.

lstrcmpi 3.0

Syntax **int lstrcmpi**(*lpString1*, *lpString2*)

This function compares the two strings identified by *lpString1* and *lpString2* lexicographically and returns a value indicating their relationship. The comparison is made based on the current language selected by the user at setup or with the Control Panel. The comparison is case-sensitive. This function is not equivalent to the **strcmpi** C run-time library function.

All strings must be less than 64K in size.

Parameter	Type/Description
lpString1	**LPSTR** Points to the first null-terminated string to be compared.
lpString2	**LPSTR** Points to the second null-terminated string to be compared.

Return Value The return value indicates whether *lpString1* is less than, equal to, or greater than *lpString2*. This relationship is outlined in the following table:

Value	Meaning
< 0	*lpString1* is less than *lpString2*.
= 0	*lpString1* is identical to *lpString2*.
> 0	*lpString1* is greater than *lpString2*.

lstrcpy

Syntax **LPSTR lstrcpy**(*lpString1*, *lpString2*)

This function copies *lpString2*, including the terminating null character, to the location specified by *lpString1*, and returns *lpString1*.

All strings must be less than 64K in size.

Parameter	Type/Description
lpString1	**LPSTR** Points to a buffer to receive the contents of *lpString2*. The buffer must be large enough to contain *lpString2*.
lpString2	**LPSTR** Points to the null-terminated string to be copied.

Return Value The return value specifies a pointer to *lpString1*. It is zero if the function fails.

lstrlen

Syntax **int lstrlen**(*lpString*)

This function returns the length, in bytes, of *lpString*, not including the terminating null character.

All strings must be less than 64K in size.

Parameter	Type/Description
lpString	**LPSTR** Points to a null-terminated string.

Return Value The return value specifies the length of *lpString*. There is no error return.

_lwrite

Syntax **int _lwrite**(*hFile*, *lpBuffer*, *wBytes*)

This function writes data into the file specified by the *hFile* parameter. The *wBytes* parameter specifies the number of bytes to write from the buffer identified by *lpBuffer*. The function return value indicates the number of bytes actually written to the file.

Parameter	Type/Description
hFile	**int** Specifies the MS-DOS file handle of the file to be read.
lpBuffer	**LPSTR** Points to a buffer that contains the data to be written to the file.
wBytes	**WORD** Specifies the number of bytes to be written to the file.

Return Value The return value indicates the number of bytes actually written to the file, or –1 if the function fails.

Comments The buffer specified by *lpBuffer* cannot extend past the end of a segment.

MAKEINTATOM

Syntax LPSTR MAKEINTATOM(*wInteger*)

This macro creates an integer atom that represents a character string of decimal digits.

Integer atoms created by this macro can be added to the atom table by means of the **Add-Atom** function.

Parameter	Type/Description
wInteger	**WORD** Specifies the numeric value of the atom's character string.

Return Value The return value points to the atom created for the given integer.

Comments The **DeleteAtom** function always succeeds for integer atoms, even though it does nothing, and the **GetAtomName** function always returns the string form of the integer atom.

MAKEINTRESOURCE

Syntax LPSTR MAKEINTRESOURCE (*nInteger*)

This macro converts an integer value into a long pointer to a string, with the high-order word of the long pointer set to zero.

Parameter	Type/Description
nInteger	**int** Specifies the integer value to be converted.

Return Value The return value points to a string.

MAKELONG

Syntax DWORD MAKELONG(*wLow*, *wHigh*)

This macro creates an unsigned long integer by concatenating two integer values, specified by the *wLow* and *wHigh* parameters.

Parameter	Type/Description
wLow	**WORD** Specifies the low-order word of the new long value.
wHigh	**WORD** Specifies the high-order word of the new long value.

Return Value The return value specifies an unsigned long-integer value.

MAKEPOINT

Syntax **POINT MAKEPOINT**(*dwInteger*)

This macro converts a long value that contains the *x*- and *y*-coordinates of a point into a **POINT** data structure.

Parameter	Type/Description
dwInteger	**DWORD** Specifies the *x*- and *y*-coordinates of a point.

Return Value The return value specifies the **POINT** data structure.

MakeProcInstance

Syntax **FARPROC MakeProcInstance**(*lpProc*, *hInstance*)

This function creates a procedure-instance address. A procedure-instance address points to prolog code that is executed before the function is executed. The prolog binds the data segment of the instance specified by the *hInstance* parameter to the function pointed to by the *lpProc* parameter. When the function is executed, it has access to variables and data in that instance's data segment.

Parameter	Type/Description
lpProc	**FARPROC** Is a procedure-instance address.
hInstance	**HANDLE** Identifies the instance associated with the desired data segment.

Return Value The return value points to the function if the function is successful. Otherwise, it is NULL.

Comments

The **MakeProcInstance** function must only be used to access functions from instances of the current module. The function is not required for library modules.

After **MakeProcInstance** has been called for a particular function, all calls to that function should be made through the retrieved address.

MakeProcInstance will create more than one procedure instance. An application should not call **MakeProcInstance** more than once using the same function and instance handle to avoid wasting memory.

To bind a data segment to a function, the function must be exported in the **EXPORTS** statement of the module-definition file.

MapDialogRect

Syntax

void MapDialogRect(*hDlg*, *lpRect*)

This function converts the dialog-box units given in the *lpRect* parameter to screen units. Dialog-box units are stated in terms of the current dialog base unit derived from the average width and height of characters in the system font. One horizontal unit is one-fourth of the dialog base width unit, and one vertical unit is one-eighth of the dialog base height unit. The **GetDialogBaseUnits** function returns the dialog base units in pixels.

The **MapDialogRect** function replaces the dialog-box units in *lpRect* with screen units (pixels), so that the rectangle can be used to create a dialog box or position a control within a box.

Parameter	Type/Description
hDlg	**HWND** Identifies a dialog box.
lpRect	**LPRECT** Points to a **RECT** data structure that contains the dialog-box coordinates to be converted.

Return Value

None.

Comments

The *hDlg* parameter must be created by using the **CreateDialog** or **DialogBox** function.

MapVirtualKey [3.0]

Syntax **WORD MapVirtualKey**(*wCode*, *wMapType*)

This function accepts a virtual-key code or scan code for a key and returns the corresponding scan code, virtual-key code, or ASCII value. The value of the *wMapType* parameter determines the type of mapping which this function performs.

Parameter	Description
wCode	**WORD** Specifies the virtual-key code or scan code for a key. The interpretation of the *wCode* parameter depends on the value of the *wMapType* parameter.
wMapType	**WORD** Specifies the type of mapping to be performed. The *wMapType* parameter can be any of the following values:

Value	Meaning
0	The *wCode* parameter specifies a virtual-key code, and the function returns the corresponding scan code.
1	The *wCode* parameter specifies a scan code, and the function returns the corresponding virtual-key code.
2	The *wCode* parameter specifies a virtual-key code, and the function returns the corresponding unshifted ASCII value.

Other values are reserved.

Return Value The return value depends on the value of the *wCode* and *wMapType* parameters. See the description of the *wMapType* parameter for more information.

max

Syntax **int max**(*value1*, *value2*)

This macro returns the greater of the values contained in the *value1* and *value2* parameters.

Parameter	Description
value1	Specifies the first of two values.
value2	Specifies the second of two values.

Return Value The return value specifies *value1* or *value2*, whichever is greater.

Comments The values identified by the *value1* and *value2* parameters can be any ordered type.

MessageBeep

Syntax **void MessageBeep**(*wType*)

This function generates a beep at the system speaker.

Parameter	Type/Description
wType	**WORD** Is not used. It should be set to zero.

Return Value None.

MessageBox

Syntax **int MessageBox**(*hWndParent*, *lpText*, *lpCaption*, *wType*)

This function creates and displays a window that contains an application-supplied message and caption, plus any combination of the predefined icons and push buttons described in the following list.

Parameter	Type/Description
hWndParent	**HWND** Identifies the window that owns the message box.
lpText	**LPSTR** Points to a null-terminated string containing the message to be displayed.
lpCaption	**LPSTR** Points to a null-terminated character string to be used for the dialog-box caption. If the *lpCaption* parameter is NULL, the default caption "Error" is used.

Parameter	Type/Description
wType	**WORD** Specifies the contents of the dialog box. It can be any combination of the values shown in Table 4.11, "Message Box Types," joined by the bitwise OR operator.

Return Value

The return value specifies the outcome of the function. It is zero if there is not enough memory to create the message box. Otherwise, it is one of the following menu-item values returned by the dialog box:

Value	Meaning
IDABORT	Abort button pressed.
IDCANCEL	Cancel button pressed.
IDIGNORE	Ignore button pressed.
IDNO	No button pressed.
IDOK	OK button pressed.
IDRETRY	Retry button pressed.
IDYES	Yes button pressed.

If a message box has a Cancel button, the IDCANCEL value will be returned if either the ESCAPE key or Cancel button is pressed. If the message box has no Cancel button, pressing the ESCAPE key has no effect.

Comments

When a system-modal message box is created to indicate that the system is low on memory, the strings passed as the *lpText* and *lpCaption* parameters should not be taken from a resource file, since an attempt to load the resource may fail.

When an application calls the **MessageBox** function and specifies the MB_ICONHAND and MB_SYSTEMMODAL flags for the *wType* parameter, Windows will display the resulting message box regardless of available memory. When these flags are specified, Windows limits the length of the message-box text to one line.

If a message box is created while a dialog box is present, use the handle of the dialog box as the *hWndParent* parameter. The *hWndParent* parameter should not identify a child window, such as a dialog-box control.

Table 4.11 shows the message box types.

Table 4.11 Message Box Types

Value	Meaning
MB_ABORTRETRYIGNORE	Message box contains three push buttons: Abort, Retry, and Ignore.
MB_APPLMODAL	The user must respond to the message box before continuing work in the window identified by the *hWndParent* parameter. However, the user can move to the windows of other applications and work in those windows. MB_APPLMODAL is the default if neither MB_SYSTEMMODAL nor MB_TASKMODAL are specified.
MB_DEFBUTTON1	First button is the default. Note that the first button is always the default unless MB_DEFBUTTON2 or MB_DEFBUTTON3 is specified.
MB_DEFBUTTON2	Second button is the default.
MB_DEFBUTTON3	Third button is the default.
MB_ICONASTERISK	Same as MB_ICONINFORMATION.
MB_ICONEXCLAMATION	An exclamation-point icon appears in the message box.
MB_ICONHAND	Same as MB_ICONSTOP.
MB_ICONINFORMATION	An icon consisting of a lowercase i in a circle appears in the message box.
MB_ICONQUESTION	A question-mark icon appears in the message box.
MB_ICONSTOP	A stop sign icon appears in the message box.
MB_OK	Message box contains one push button: OK.
MB_OKCANCEL	Message box contains two push buttons: OK and Cancel.
MB_RETRYCANCEL	Message box contains two push buttons: Retry and Cancel.
MB_SYSTEMMODAL	All applications are suspended until the user responds to the message box. Unless the application specifies MB_ICONHAND, the message box does not become modal until after it is created; consequently, the parent window and other windows continue to receive messages resulting from its activation. System-modal message boxes are used to notify the user of serious, potentially damaging errors that require immediate attention (for example, running out of memory).

Table 4.11 Message Box Types *(continued)*

Value	Meaning
MB_TASKMODAL	Same as MB_APPMODAL except that all the top-level windows belonging to the current task are disabled if the *hWndOwner* parameter is NULL. This flag should be used when the calling application or library does not have a window handle available, but still needs to prevent input to other windows in the current application without suspending other applications.
MB_YESNO	Message box contains two push buttons: Yes and No.
MB_YESNOCANCEL	Message box contains three push buttons: Yes, No, and Cancel.

min

Syntax

int min(*value1*, *value2*)

This macro returns the lesser of the values specified by the *value1* and *value2* parameters, respectively.

Parameter	Description
value1	Specifies the first of two values.
value2	Specifies the second of two values.

Return Value

The return value specifies *value1* or *value2*, whichever is less.

Comments

The values identified by the *value1* and *value2* parameters can be any ordered type.

ModifyMenu 3.0

Syntax

BOOL ModifyMenu(*hMenu*, *nPosition*, *wFlags*, *wIDNewItem*, *lpNewItem*)

This function changes an existing menu item at the position specified by the *nPosition* parameter. The application specifies the new state of the menu item by setting values in the *wFlags* parameter. If this function replaces a pop-up menu associated with the menu item, it destroys the old pop-up menu and frees the memory used by the pop-up menu.

Parameter	Type/Description
hMenu	**HMENU** Identifies the menu to be changed.
nPosition	**WORD** Specifies the menu item to be changed. The interpretation of the *nPosition* parameter depends upon the setting of the *wFlags* parameter.

If *wFlags* is:	*nPosition*
MF_BYPOSITION	Specifies the position of the existing menu item. The first item in the menu is at position zero.
MF_BYCOMMAND	Specifies the command ID of the existing menu item.

wFlags	**WORD** Specifies how the *nPosition* parameter is interpreted and information about the changes to be made to the menu item. It consists of one or more values listed in the following "Comments" section.
wIDNewItem	**WORD** Specifies either the command ID of the modified menu item or, if *wFlags* is set to MF_POPUP, the menu handle of the pop-up menu.
lpNewItem	**LPSTR** Specifies the content of the changed menu item. If *wFlags* is set to MF_STRING (the default), then *lpNewItem* is a long pointer to a null-terminated character string. If *wFlags* is set to MF_BITMAP instead, then *lpNewItem* contains a bitmap handle (**HBITMAP**) in its low-order word. If *wFlags* is set to MF_OWNERDRAW, *lpNewItem* specifies an application-supplied 32-bit value which the application can use to maintain additional data associated with the menu item. This 32-bit value is available to the application in the **itemData** field of the structure, pointed to by the *lParam* parameter of the following messages:

WM_MEASUREITEM
WM_DRAWITEM

These messages are sent when the menu item is initially displayed, or is changed.

Return Value The return value specifies the outcome of the function. It is TRUE if the function is successful. Otherwise, it is FALSE.

Comments

Whenever a menu changes (whether or not the menu resides in a window that is displayed), the application should call **DrawMenuBar**. In order to change the attributes of existing menu items, it is much faster to use the **CheckMenuItem** and **EnableMenuItem** functions.

Each of the following groups lists flags that should not be used together:

- MF_BYCOMMAND and MF_BYPOSITION
- MF_DISABLED, MF_ENABLED, and MF_GRAYED
- MF_BITMAP, MF_STRING, MF_OWNERDRAW, and MF_SEPARATOR
- MF_MENUBARBREAK and MF_MENUBREAK
- MF_CHECKED and MF_UNCHECKED

The following list describes the flags which may be set in the *wFlags* parameter:

Value	Meaning
MF_BITMAP	Uses a bitmap as the menu item. The low-order word of the *lpNewItem* parameter contains the handle of the bitmap.
MF_BYCOMMAND	Specifies that the *nPosition* parameter gives the menu item control ID number. This is the default if neither MF_BYCOMMAND nor MF_POSITION is set.
MF_BYPOSITION	Specifies that the *nPosition* parameter gives the position of the menu item to be changed rather than an ID number.
MF_CHECKED	Places a checkmark next to the menu item. If the application has supplied checkmark bitmaps (see **SetMenuItemBitmaps**), setting this flag displays the "checkmark on" bitmap next to the menu item.
MF_DISABLED	Disables the menu item so that it cannot be selected, but does not gray it.
MF_ENABLED	Enables the menu item so that it can be selected and restores it from its grayed state.
MF_GRAYED	Disables the menu item so that it cannot be selected and grays it.
MF_MENUBARBREAK	Same as MF_MENUBREAK except that for pop-up menus, separates the new column from the old column with a vertical line.

Value	Meaning
MF_MENUBREAK	Places the menu item on a new line for static menu-bar items. For pop-up menus, this flag places the item in a new column, with no dividing line between the columns.
MF_OWNERDRAW	Specifies that the menu item is an owner-draw item. The window that owns the menu receives a WM_MEASURE-ITEM message when the menu is displayed for the first time to retrieve the height and width of the menu item. The WM_DRAWITEM message is then sent whenever the owner must update the visual appearance of the menu item. This option is not valid for a top-level menu item.
MF_POPUP	Specifies that the item has a pop-up menu associated with it. The *wIDNewItem* parameter specifies a handle to a pop-up menu to be associated with the menu item. Use this flag for adding either a top-level pop-up menu or adding a hierarchical pop-up menu to a pop-up menu item.
MF_SEPARATOR	Draws a horizontal dividing line. You can only use this flag in a pop-up menu. This line cannot be grayed, disabled, or highlighted. The *lpNewItem* and *wIDNewItem* parameters are ignored.
MF_STRING	Specifies that the menu item is a character string; the *lpNewItem* parameter points to the string for the menu item.
MF_UNCHECKED	Does not place a checkmark next to the menu item. No checkmark is the default if neither MF_CHECKED nor MF_UNCHECKED is set. If the application has supplied checkmark bitmaps (see **SetMenuItemBitmaps**), setting this flag displays the "checkmark off" bitmap next to the menu item.

MoveTo

Syntax **DWORD** **MoveTo**(*hDC*, *X*, *Y*)

This function moves the current position to the point specified by the *X* and *Y* parameters.

Parameter	Type/Description
hDC	**HDC** Identifies the device context.
X	**int** Specifies the logical *x*-coordinate of the new position.
Y	**int** Specifies the logical *y*-coordinate of the new position.

Return Value The return value specifies the *x*- and *y*-coordinates of the previous position. The *y*-coordinate is in the high-order word; the *x*-coordinate is in the low-order word.

Comments Although the **MoveTo** function has no output, it affects other output functions that use the current position.

MoveWindow

Syntax **void MoveWindow**(*hWnd*, *X*, *Y*, *nWidth*, *nHeight*, *bRepaint*)

This function causes a WM_SIZE message to be sent to the given window. The *X*, *Y*, *nWidth*, and *nHeight* parameters give the new size of the window.

Parameter	Type/Description
hWnd	**HWND** Identifies a pop-up or child window.
X	**int** Specifies the new *x*-coordinate of the upper-left corner of the window.
Y	**int** Specifies the new *y*-coordinate of the upper-left corner of the window. For pop-up windows, *X* and *Y* are in screen coordinates (relative to the upper-left corner of the screen). For child windows, they are in client coordinates (relative to the upper-left corner of the parent window's client area).
nWidth	**int** Specifies the new width of the window.
nHeight	**int** Specifies the new height of the window.
bRepaint	**BOOL** Specifies whether or not the window is repainted after moving. If *bRepaint* is zero, the window is not repainted.

Return Value None.

Comments

Any child or pop-up window has a minimum width and height. These minimums depend on the style and content of the window. Any attempt to make the width and height smaller than the minimum by using the **MoveWindow** function will fail. The WM_SIZE message created by this function gives the new width and height of the client area of the window, not of the full window.

MulDiv 3.0

Syntax

int MulDiv(*nNumber*, *nNumerator*, *nDenominator*)

This function multiplies two word-length values and then divides the result by a third word-length value. The return value is the final result, rounded to the nearest integer.

Parameter	Type/Description
nNumber	**int** Specifies the number to be multiplied by *nNumerator*.
nNumerator	**int** Specifies the number to be multiplied by *nNumber*.
nDenominator	**int** Specifies the number by which the result of the multiplication is to be divided.

Return Value

The return value is the result of the multipliation and division. The return value is 32,767 or –32,767 if either an overflow occurred or *wDenominator* was zero.

NetBIOSCall 3.0

This function allows an applications to issue the NETBIOS interrupt 5CH. An application should call this function instead of directly issuing a NETBIOS 5CH interrupt to preserve compatibility with future Microsoft products.

An application can call this function only from an assembly-language routine. It is exported from KERNEL.EXE and is not defined in any Windows include files.

To use this function call, an application should declare it in an assembly-language program as shown:

```
extrn    NETBIOSCALL    :far
```

If the application includes CMACROS.INC, the application declares it as shown:

```
externFP  NetBIOSCall
```

Before calling **NetBIOSCall**, all registers must be set as for an actual INT 5CH. All registers at the function's exit are the same as for the corresponding INT 5CH function.

This function has no parameters and no return value.

The following is an example of how to use the **NetBIOSCall** function:

```
extrn NETBIOSCALL : far
        .
        .
        .
;set registers
cCall NetBIOSCall
```

OemKeyScan 3.0

Syntax **DWORD OemKeyScan**(*wOemChar*)

This function maps OEM ASCII codes 0 through 0x0FF into the OEM scan codes and shift states. It provides information which allows a program to send OEM text to another program by simulating keyboard input and is used specifically for this purpose by Windows in 386 enhanced mode.

Parameter	Type/Description
wOemChar	**WORD** Specifies the ASCII value of the OEM character.

Return Value The return value contains in its low-order word the scan code of the OEM character identified by the *wOemChar* parameter. The high-order word of the return value contains flags which indicate the shift state. The following lists the flag bits and their meanings:

Bit	Meaning
2	CTRL key is pressed.
1	Either SHIFT key is pressed.

If the character is not defined in the OEM character tables, both the low-order and high-order words of the return value contain –1.

Comments This function does not provide translations for characters which require CTRL-ALT or dead keys. Characters not translated by this function must be copied by simulating input using the "ALT + keypad" mechanism. The NUMLOCK key must be off.

This function calls the **VkKeyScan** function in recent versions of the keyboard drivers.

OemToAnsi

Syntax **int OemToAnsi**(*lpOemStr*, *lpAnsiStr*)

This function translates the string pointed to by the *lpOemStr* parameter from the OEM-defined character set into the ANSI character set. The string can be greater than 64K in length.

Parameter	Type/Description
lpOemStr	**LPSTR** Points to a null-terminated string of characters from the OEM-defined character set.
lpAnsiStr	**LPSTR** Points to the location where the translated string is to be copied. The *lpAnsiStr* parameter can be the same as *lpOemStr* to translate the string in place.

Return Value The return value is always –1.

OemToAnsiBuff

Syntax void **OemToAnsiBuff**(*lpOemStr*, *lpAnsiStr*, *nLength*)

This function translates the string in the buffer pointed to by the *lpOemStr* parameter from the OEM-defined character set into the ANSI character set.

Parameter	Type/Description
lpOemStr	**LPSTR** Points to a buffer containing one or more characters from the OEM-defined character set.
lpAnsiStr	**LPSTR** Points to the location where the translated string is to be copied. The *lpAnsiStr* parameter can be the same as *lpOemStr* to translate the string in place.
nLength	**WORD** Specifies the number of characters in the buffer identified by the *lpOemStr* parameter. If *nLength* is zero, the length is 64K (65,536).

Return Value None.

OffsetClipRgn

Syntax int **OffsetClipRgn**(*hDC*, *X*, *Y*)

This function moves the clipping region of the given device by the specified offsets. The function moves the region *X* units along the *x*-axis and *Y* units along the *y*-axis.

Parameter	Type/Description
hDC	**HDC** Identifies the device context.
X	**int** Specifies the number of logical units to move left or right.
Y	**int** Specifies the number of logical units to move up or down.

Return Value The return value specifies the new region's type. It can be any one of the following values:

Value	Meaning
COMPLEXREGION	Clipping region has overlapping borders.
ERROR	Device context is not valid.
NULLREGION	Clipping region is empty.
SIMPLEREGION	Clipping region has no overlapping borders.

OffsetRect

Syntax **void OffsetRect**(*lpRect*, *X*, *Y*)

This function moves the given rectangle by the specified offsets. The **OffsetRect** function moves the rectangle *X* units along the *x*-axis and *Y* units along the *y*-axis. The *X* and *Y* parameters are signed values, so the rectangle can be moved left or right, and up or down.

Parameter	Type/Description
lpRect	**LPRECT** Points to a **RECT** data structure that contains the rectangle to be moved.
X	**int** Specifies the amount to move left or right. It must be negative to move left.
Y	**int** Specifies the amount to move up or down. It must be negative to move up.

Return Value None.

Comments The coordinate values of a rectangle must not be greater than 32,767 or less than –32,768. The *X* and *Y* parameters must be chosen carefully to prevent invalid rectangles.

OffsetRgn

Syntax **int OffsetRgn**(*hRgn*, *X*, *Y*)

This function moves the given region by the specified offsets. The function moves the region *X* units along the *x*-axis and *Y* units along the *y*-axis.

Parameter	Type/Description
hRgn	**HRGN** Identifies the region to be moved.
X	**int** Specifies the number of units to move left or right.
Y	**int** Specifies the number of units to move up or down.

Return Value The return value specifies the new region's type. It can be any one of the following values:

Value	Meaning
COMPLEXREGION	Region has overlapping borders.
ERROR	Region handle is not valid.
NULLREGION	Region is empty.
SIMPLEREGION	Region has no overlapping borders.

Comments The coordinate values of a region must not be greater than 32,767 or less than –32,768. The *X* and *Y* parameters must be carefully chosen to prevent invalid regions.

OffsetViewportOrg

Syntax **DWORD OffsetViewportOrg**(*hDC*, *X*, *Y*)

This function modifies the viewport origin relative to the current values. The formulas are written as follows:

xNewVO = xOldVO + X
yNewVO = yOldVO + Y

The new origin is the sum of the current origin and the *X* and *Y* values.

Parameter	Type/Description
hDC	**HDC** Identifies the device context.
X	**int** Specifies the number of device units to add to the current origin's *x*-coordinate.
Y	**int** Specifies the number of device units to add to the current origin's *y*-coordinate.

Return Value The return value specifies the previous viewport origin (in device coordinates). The previous *y*-coordinate is in the high-order word; the previous *x*-coordinate is in the low-order word.

OffsetWindowOrg

Syntax **DWORD OffsetWindowOrg**(*hDC*, *X*, *Y*)

This function modifies the viewport origin relative to the current values. The formulas are written as follows:

xNewWO = xOldWO + X
yNewWO = yOldWO + Y

The new origin is the sum of the current origin and the *X* and *Y* values.

Parameter	Type/Description
hDC	**HDC** Identifies the device context.
X	**int** Specifies the number of logical units to add to the current origin's *x*-coordinate.
Y	**int** Specifies the number of logical units to add to the current origin's *y*-coordinate.

Return Value The return value specifies the previous window origin (in logical coordinates). The previous *y*-coordinate is in the high-order word; the previous *x*-coordinate is in the low-order word.

OpenClipboard

Syntax **BOOL OpenClipboard**(*hWnd*)

This function opens the clipboard for examination and prevents other applications from modifying the clipboard contents.

Parameter	Type/Description
hWnd	**HWND** Identifies the window to be associated with the open clipboard.

Return Value The return value specifies the status of the clipboard. It is nonzero if the clipboard is opened. If the clipboard has already been opened by another application, the return value is zero.

Comments An application should call the **CloseClipboard** function for every successful call to the **OpenClipboard** function.

OpenComm

Syntax **int OpenComm**(*lpComName*, *wInQueue*, *wOutQueue*)

This function opens a communication device and assigns an *nCid* handle to it. The communication device is initialized to a default configuration. The **SetCommState** function should be used to initialize the device to alternate values. The **OpenComm** function allocates space for receive and transmit queues. The queues are used by the interrupt-driven transmit/receive software.

Parameter	Type/Description
lpComName	**LPSTR** Points to a string which contains COM*n* or LPT*n*, where *n* ranges from 1 to the number of communication devices available for the particular type of I/O port.
wInQueue	**WORD** Specifies the size of the receive queue.
wOutQueue	**WORD** Specifies the size of the transmit queue.

Return Value The return value specifies the open communication device. If an error occurs, the return value is one of the following negative error values:

Value	Meaning
IE_BADID	Invalid or unsupported ID.
IE_BAUDRATE	Unsupported baud rate.
IE_BYTESIZE	Invalid byte size.
IE_DEFAULT	Error in default parameters.
IE_HARDWARE	Hardware not present.
IE_MEMORY	Unable to allocate queues.
IE_NOPEN	Device not open.
IE_OPEN	Device already open.

Comments LPT ports are not interrupt driven. For these ports, the *nInQueue* and *nOutQueue* parameters are ignored, and the queue size is set to zero.

OpenFile

Syntax **int OpenFile**(*lpFileName*, *lpReOpenBuff*, *wStyle*)

This function creates, opens, reopens, or deletes a file.

Parameter	Type/Description
lpFileName	**LPSTR** Points to a null-terminated character string that names the file to be opened. The string must consist of characters from the ANSI character set.
lpReOpenBuff	**LPOFSTRUCT** Points to the **OFSTRUCT** data structure that is to receive information about the file when the file is first opened. The structure can be used in subsequent calls to the **OpenFile** function to refer to the open file. The **szPathName** field of this data structure contains characters from the OEM character set.
wStyle	**WORD** Specifies the action to take. These styles can be combined by using the bitwise OR operator:

Parameter	Type/Description	
	Value	**Meaning**
	OF_CANCEL	Adds a Cancel button to the OF_PROMPT dialog box. Pressing the Cancel button directs **OpenFile** to return a file-not-found error message.
	OF_CREATE	Directs **OpenFile** to create a new file. If the file already exists, it is truncated to zero length.
	OF_DELETE	Deletes the file.
	OF_EXIST	Opens the file, and then closes it. Used to test for file existence.
	OF_PARSE	Fills the **OFSTRUCT** data structure but carries out no other action.
	OF_PROMPT	Displays a dialog box if the requested file does not exist. The dialog box informs the user that Windows cannot find the file and prompts the user to insert the file in drive A.
	OF_READ	Opens the file for reading only.
	OF_READWRITE	Opens the file for reading and writing.
	OF_REOPEN	Opens the file using information in the re-open buffer.

Parameter	Type/Description	
	Value	**Meaning**
	OF_SHARE_COMPAT	Opens the file with compatibility mode, allowing any process on a given machine to open the file any number of times. **OpenFile** fails if the file has been opened with any of the other sharing modes.
	OF_SHARE_DENY_NONE	Opens the file without denying other processes read or write access to the file. **OpenFile** fails if the file has been opened in compatibility mode by any other process.
	OF_SHARE_DENY_READ	Opens the file and denies other processes read access to the file. **OpenFile** fails if the file has been opened in compatibility mode or for read access by any other process.
	OF_SHARE_DENY_WRITE	Opens the file and denies other processes write access to the file. **OpenFile** fails if the file has been opened in compatibility or for write access by any other process.
	OF_SHARE_EXCLUSIVE	Opens the file with exclusive mode, denying other processes both read and write access to the file. **OpenFile** fails if the file has been opened in any other mode for read or write access, even by the current process.

Parameter	Type/Description	
	Value	**Meaning**
	OF_VERIFY	Verifies that the date and time of the file are the same as when it was previously opened. Useful as an extra check for read-only files.
	OF_WRITE	Opens the file for writing only.

Return Value

The return value specifies a DOS file handle if the function is successful. Otherwise, it is –1.

Comments

If the *lpFileName* parameter specifies a filename and extension only, this function searches for a matching file in the following directories:

1. The current directory.
2. The Windows directory (the directory containing WIN.COM); the **GetWindowsDirectory** function obtains the pathname of this directory.
3. The Windows system directory (the directory containing such system files as KERNEL.EXE); the **GetSystemDirectory** function obtains the pathname of this directory.
4. Any of the directories listed in the PATH environment variable.
5. Any directory in the list of directories mapped in a network.

Windows searches the directories in the listed order.

The *lpFileName* parameter cannot contain wildcard characters.

To close the file after use, the application should call the **_lclose** function.

OpenIcon

Syntax

BOOL OpenIcon(*hWnd*)

This function activates and displays an iconic (minimized) window. Windows restores it to its original size and position.

Parameter	Type/Description
hWnd	**HWND** Identifies the window.

Return Value The return value specifies the outcome of the function. It is nonzero if the function is successful. Otherwise, it is zero.

OpenSound

Syntax **int OpenSound( )**

This function accesses the play device and prevents it from being opened subsequently by other applications.

This function has no parameters.

Return Value The return value specifies the number of voices available. The return value is S_SERDVNA if the play device is in use, and S_SEROFM if insufficient memory is available.

OutputDebugString 3.0

Syntax **void OutputDebugString**(*lpOutputString*)

This function sends a debugging message to the debugger if present, or to the auxiliary (AUX) device if the debugger is not present.

Parameter	Type/Description
lpOutputString	**LPSTR** Points to a null-terminated string.

Return Value None.

Comments This function preserves all registers. It is available only in the debugging version of Windows.

PaintRgn

Syntax **BOOL** **PaintRgn**(*hDC*, *hRgn*)

This function fills the region specified by the *hRgn* parameter with the selected brush.

Parameter	Type/Description
hDC	**HDC** Identifies the device context that contains the region.
hRgn	**HRGN** Identifies the region to be filled. The coordinates for the given region are specified in device units.

Return Value The return value specifies the outcome of the function. It is nonzero if the function is successful. Otherwise, it is zero.

PALETTEINDEX 3.0

Syntax **COLORREF** **PALETTEINDEX**(*nPaletteIndex*)

This macro accepts an index to a logical color palette entry and returns a value consisting of 1 in the high-order byte and the palette entry index in the low-order bytes. This is called a palette-entry specifier. An application using a color palette can pass this specifier instead of an explicit RGB value to functions that expect a color. This allows the function to use the color in the specified palette entry.

Parameter	Type/Description
nPaletteIndex	**int** Specifies an index to the palette entry containing the color to be used for a graphics operation.

Return Value The return value is a logical-palette index specifier. When using a logical palette, an application can use this specifier in place of an explicit RGB value for GDI functions that require a color.

PALETTERGB 3.0

Syntax **COLORREF** **PALETTERGB**(*cRed*, *cGreen*, *cBlue*)

This macro accepts three values representing relative intensities of red, green, and blue, and returns a value consisting of 2 in the high-order byte and an RGB value in the three low-order bytes. This is called a palette-relative RGB specifier. An application using a color palette can pass this specifier instead of an explicit RGB value to functions that expect a color.

For output devices that support logical palettes, Windows matches a palette-relative RGB value to the nearest color in the logical palette of the device context, as though the application had specified an index to that palette entry. If an output device does not support a system palette, then Windows uses the palette-relative RGB as though it were a conventional RGB **DWORD** returned by the **RGB** macro.

Parameter	Type/Description
cRed	**BYTE** Specifies the intensity of the red color field.
cGreen	**BYTE** Specifies the intensity of the green color field.
cBlue	**BYTE** Specifies the intensity of the blue color field.

Return Value The return value specifies a palette-relative RGB value.

PatBlt

Syntax **BOOL PatBlt**(*hDC*, *X*, *Y*, *nWidth*, *nHeight*, *dwRop*)

This function creates a bit pattern on the specified device. The pattern is a combination of the selected brush and the pattern already on the device. The raster-operation code specified by the *dwRop* parameter defines how the patterns are to be combined.

Parameter	Type/Description
hDC	**HDC** Identifies the device context.
X	**int** Specifies the logical *x*-coordinate of the upper-left corner of the rectangle that is to receive the pattern.
Y	**int** Specifies the logical *y*-coordinate of the upper-left corner of the rectangle that is to receive the pattern.
nWidth	**int** Specifies the width (in logical units) of the rectangle that is to receive the pattern.
nHeight	**int** Specifies the height (in logical units) of the rectangle that is to receive the pattern.

Parameter	Type/Description
dwRop	**DWORD** Specifies the raster-operation code. Raster-operation codes (ROPs) define how GDI combines colors in output operations that involve a current brush, a possible source bitmap, and a destination bitmap. For a list of the raster-operation codes, see Table 4.12, "Raster Operations."

Return Value The return value specifies the outcome of the function. It is nonzero if the bit pattern is drawn. Otherwise, it is zero.

Comments The values of *dwRop* for this function are a limited subset of the full 256 ternary raster-operation codes; in particular, an operation code that refers to a source cannot be used.

Not all devices support the **PatBlt** function. For more information, see the RC_BITBLT capability in the **GetDeviceCaps** function, earlier in this chapter.

Table 4.12 lists the various raster-operation codes for the *dwRop* parameter:

Table 4.12 Raster Operations

Code	Description
PATCOPY	Copies pattern to destination bitmap.
PATINVERT	Combines destination bitmap with pattern using the Boolean OR operator.
DSTINVERT	Inverts the destination bitmap.
BLACKNESS	Turns all output black.
WHITENESS	Turns all output white.

PeekMessage

Syntax **BOOL PeekMessage**(*lpMsg*, *hWnd*, *wMsgFilterMin*, *wMsgFilterMax*, *wRemoveMsg*)

This function checks the application queue for a message and places the message (if any) in the data structure pointed to by the *lpMsg* parameter. Unlike the **GetMessage** function, the **PeekMessage** function does not wait for a message to be placed in the queue before returning. It does, however, yield control (if the PM_NOYIELD flag isn't set) and does not return control after the yield until Windows returns control to the application.

PeekMessage retrieves only messages associated with the window specified by the *hWnd* parameter, or any of its children as specified by the **IsChild** function, and within the range of message values given by the *wMsgFilterMin* and *wMsgFilterMax* parameters. If *hWnd* is NULL, **PeekMessage** retrieves messages for any window that belongs to the application making the call. (The **PeekMessage** function does not retrieve messages for windows that belong to other applications.) If *hWnd* is –1, **PeekMessage** returns only messages with a *hWnd* of NULL as posted by the **PostAppMessage** function. If *wMsgFilterMin* and *wMsgFilterMax* are both zero, **PeekMessage** returns all available messages (no range filtering is performed).

The WM_KEYFIRST and WM_KEYLAST flags can be used as filter values to retrieve all key messages; the WM_MOUSEFIRST and WM_MOUSELAST flags can be used to retrieve all mouse messages.

Parameter	Type/Description
lpMsg	**LPMSG** Points to an **MSG** data structure that contains message information from the Windows application queue.
hWnd	**HWND** Identifies the window whose messages are to be examined.
wMsgFilterMin	**WORD** Specifies the value of the lowest message position to be examined.
wMsgFilterMax	**WORD** Specifies the value of the highest message position to be examined.
wRemoveMsg	**WORD** Specifies a combination of the flags described in the following list. PM_NOYIELD can be combined with either PM_NOREMOVE or PM_REMOVE:

Value	Meaning
PM_NOREMOVE	Messages are not removed from the queue after processing by **PeekMessage.**
PM_NOYIELD	Prevents the current task from halting and yielding system resources to another task.
PM_REMOVE	Messages are removed from the queue after processing by **PeekMessage.**

Return Value

The return value specifies whether or not a message is found. It is nonzero if a message is available. Otherwise, it is zero.

Comments

PeekMessage does not remove WM_PAINT messages from the queue. The messages remain in the queue until processed. The **GetMessage, PeekMessage,** and **WaitMessage** functions yield control to other applications. These calls are the only way to let other applications run. If your application does not call any of these functions for long periods of time, other applications cannot run.

When **GetMessage**, **PeekMessage**, and **WaitMessage** yield control to other applications, the stack and data segments of the application calling the function may move in memory to accommodate the changing memory requirements of other applications.

If the application has stored long pointers to objects in the data or stack segment (global or local variables), and if they are unlocked, these pointers can become invalid after a call to **GetMessage**, **PeekMessage**, or **WaitMessage**. The *lpMsg* parameter of the called function remains valid in any case.

Pie

Syntax

BOOL **Pie**(*hDC*, *X1*, *Y1*, *X2*, *Y2*, *X3*, *Y3*, *X4*, *Y4*)

This function draws a pie-shaped wedge by drawing an elliptical arc whose center and two endpoints are joined by lines. The center of the arc is the center of the bounding rectangle specified by the *X1*, *Y1*, *X2*, and *Y2* parameters. The starting and ending points of the arc are specified by the *X3*, *Y3*, *X4*, and *Y4* parameters. The arc is drawn with the selected pen, moving in a counterclockwise direction. Two additional lines are drawn from each endpoint to the arc's center. The pie-shaped area is filled with the selected brush.

If *X3* equals *X4* and *Y3* equals *Y4*, the result is an ellipse with a single line from the center of the ellipse to the point (*X3*, *Y3*), or (*X4*, *Y4*).

Parameter	Type/Description
hDC	**HDC** Identifies the device context.
X1	**int** Specifies the logical *x*-coordinate of the upper-left corner of the bounding rectangle.
Y1	**int** Specifies the logical *y*-coordinate of the upper-left corner of the bounding rectangle.
X2	**int** Specifies the logical *x*-coordinate of the lower-right corner of the bounding rectangle.
Y2	**int** Specifies the logical *y*-coordinate of the lower-right corner of the bounding rectangle.
X3	**int** Specifies the logical *x*-coordinate of the starting point of the arc. This point does not have to lie exactly on the arc.
Y3	**int** Specifies the logical *y*-coordinate of the starting point of the arc. This point does not have to lie exactly on the arc.

Parameter	Type/Description
X4	**int** Specifies the logical *x*-coordinate of the endpoint of the arc. This point does not have to lie exactly on the arc.
Y4	**int** Specifies the logical *y*-coordinate of the endpoint of the arc. This point does not have to lie exactly on the arc.

Return Value The return value specifies whether or not the pie shape is drawn. It is nonzero if the pie shape is drawn. Otherwise, it is zero.

Comments The width of the rectangle, specified by the absolute value of $X2 - X1$, must not exceed 32,767 units. This limit applies to the height of the rectangle as well.

The current position is neither used nor updated by this function.

PlayMetaFile

Syntax **BOOL PlayMetaFile**(*hDC*, *hMF*)

This function plays the contents of the specified metafile on the given device. The metafile can be played any number of times.

Parameter	Type/Description
hDC	**HDC** Identifies the device context of the output device.
hMF	**HANDLE** Identifies the metafile.

Return Value The return value specifies the outcome of the function. It is nonzero if the function is successful. Otherwise, it is zero.

PlayMetaFileRecord

Syntax **void PlayMetaFileRecord**(*hDC*, *lpHandletable*, *lpMetaRecord*, *nHandles*)

This function plays a metafile record by executing the GDI function call contained within the metafile record.

Parameter	Type/Description
hDC	**HDC** Identifies the device context of the output device.
lpHandletable	**LPHANDLETABLE** Points to the object handle table to be used for the metafile playback.
lpMetaRecord	**LPMETARECORD** Points to the metafile to be played.
nHandles	**WORD** Specifies the number of handles in the handle table.

Return Value None.

Comments An application typically uses this function in conjunction with the **EnumMetafile** function to modify and then play a metafile.

Polygon

Syntax **BOOL Polygon**(*hDC*, *lpPoints*, *nCount*)

This function draws a polygon consisting of two or more points (vertices) connected by lines. The polygons are filled using the current polygon-filling mode. For a description of the polygon-filling mode, see the **SetPolyFillMode** function, later in this chapter. The polygon is automatically closed, if necessary, by drawing a line from the last vertex to the first.

Parameter	Type/Description
hDC	**HDC** Identifies the device context.
lpPoints	**LPPOINT** Points to an array of points that specify the vertices of the polygon. Each point in the array is a **POINT** data structure.
nCount	**int** Specifies the number of vertices given in the array.

Return Value The return value specifies the outcome of the function. It is nonzero if the function is successful. Otherwise, it is zero.

Comments The current position is neither used nor updated by this function.

The current polygon-filling mode can be retrieved or set by using the **GetPolyFillMode** and **SetPolyFillMode** functions.

Polyline

Syntax **BOOL** **Polyline**(*hDC*, *lpPoints*, *nCount*)

This function draws a set of line segments, connecting the points specified by the *lpPoints* parameter. The lines are drawn from the first point through subsequent points with the result as if the **MoveTo** and **LineTo** functions were used to move to each new point and then connect it to the next. However, the current position is neither used nor updated by the **Polyline** function.

Parameter	Type/Description
hDC	**HDC** Identifies the device context.
lpPoints	**LPPOINT** Points to an array of points to be connected. Each point in the array is a **POINT** data structure.
nCount	**int** Specifies the number of points in the array. The *nCount* parameter must be at least 2.

Return Value The return value specifies whether or not the line segments were drawn. It is nonzero if the line segments were drawn. Otherwise, it is zero.

Comments This function draws lines with the selected pen.

PolyPolygon 3.0

Syntax **BOOL** **PolyPolygon**(*hDC*, *lpPoints*, *lpPolyCounts*, *nCount*)

This function creates a series of closed polygons. The polygons are filled using the current polygon-filling mode. For a description of the polygon-filling mode, see the **SetPolyFillMode** function, later in this chapter. The polygons may overlap, but they do not have to overlap.

Parameter	Description
hDC	**HDC** Identifies the device context.
lpPoints	**LPPOINT** Points to an array of **POINT** data structures that define the vertices of the polygons. Each polygon must be a closed polygon. Unlike polygons created by the **Polygon** function, the polygons created by **PolyPolygon** are not automatically closed. The polygons are specified consecutively.

Parameter	Description
lpPolyCounts	**LPINT** Points to an array of integers, each of which specifies the number of points in one of the polygons in the *lpPoints* array.
nCount	**int** Specifies the total number of integers in the *lpPolyCounts* array.

Return Value The return value specifies the outcome of the function. It is nonzero if the polygons were drawn. Otherwise, it is zero.

PostAppMessage

Syntax **BOOL PostAppMessage**(*hTask*, *wMsg*, *wParam*, *lParam*)

This function posts a message to an application identified by a task handle, and then returns without waiting for the application to process the message. The application receiving the message obtains the message by calling the **GetMessage** or **PeekMessage** function. The *hWnd* parameter of the returned **MSG** structure is NULL.

Parameter	Type/Description
hTask	**HANDLE** Identifies the task that is to receive the message. The **GetCurrentTask** function returns this handle.
wMsg	**WORD** Specifies the type of message posted.
wParam	**WORD** Specifies additional message information.
lParam	**DWORD** Specifies additional message information.

Return Value The return value specifies whether or not the message is posted. It is nonzero if the message is posted. Otherwise, it is zero.

PostMessage

Syntax **BOOL PostMessage**(*hWnd*, *wMsg*, *wParam*, *lParam*)

This function places a message in a window's application queue, and then returns without waiting for the corresponding window to process the message. The posted message can be retrieved by calls to the **GetMessage** or **PeekMessage** function.

Parameter	Type/Description
hWnd	**HWND** Identifies the window to receive the message. If the *hWnd* parameter is 0xFFFF, the message is sent to all overlapped or pop-up windows in the system. The message is not sent to child windows.
wMsg	**WORD** Specifies the type of message posted.
wParam	**WORD** Specifies additional message information.
lParam	**DWORD** Specifies additional message information.

Return Value The return value specifies whether or not the message is posted. It is nonzero if the message is posted. Otherwise, it is zero.

Comments An application should never use the **PostMessage** function to send a message to a control. If a system running Windows is configured for an expanded-memory system (EMS) and an application sends a message (by using the **PostMessage** function) with related data (that are pointed to by the *lParam* parameter) to a second application, the first application must place the data (that *lParam* points to) in global memory allocated with the **GlobalAlloc** function and the **GMEM_LOWER** flag. Note that this allocation of memory is necessary only if *lParam* contains a pointer.

Unlike other Windows functions, an application may call **PostMessage** at the hardware-interrupt level.

PostQuitMessage

Syntax **void PostQuitMessage**(*nExitCode*)

This function informs Windows that the application wishes to terminate execution. It is typically used in response to a WM_DESTROY message.

The **PostQuitMessage** function posts a WM_QUIT message to the application and returns immediately; the function merely informs the system that the application wants to quit sometime in the future.

When the application receives the WM_QUIT message, it should exit the message loop in the main function and return control to Windows. The exit code returned to Windows must be the *wParam* parameter of the WM_QUIT message.

Parameter	Type/Description
nExitCode	**int** Specifies an application exit code. It is used as the *wParam* parameter of the WM_QUIT message.

Return Value None.

ProfClear 3.0

Syntax **void ProfClear( )**

When running the Microsoft Windows Profiler, this function discards all samples currently in the sampling buffer. See *Tools* for more information on using the Profiler.

This function has no parameters.

Return Value None.

ProfFinish 3.0

Syntax **void ProfFinish( )**

When running the Microsoft Windows Profiler, this function stops sampling and flushes the output buffer to disk.

When running with Windows in 386 enhanced mode, **ProfFinish** also frees the buffer for system use. See *Tools* for more information on using the Profiler.

This function has no parameters.

Return Value None.

ProfFlush 3.0

Syntax **void ProfFlush( )**

When running the Microsoft Windows Profiler, this function flushes the sampling buffer to disk, provided that samples do not exceed predefined limits.

When running with Windows in any mode other than 386 enhanced mode, you must specify the size of the output buffer and the amount of samples to be written to disk.

When running with Windows in 386 enhanced mode, an application calls the **ProfSetup** function to specify the size of the output buffer and the amount of samples to be written to disk.

See *Tools* for more information on using the Profiler.

This function has no parameters.

Return Value None.

Comments Do not call **ProfFlush** repeatedly because it can seriously impair the performance of the application. Additionally, do not call the function when DOS may be unstable, as in interrupt handling.

ProfInsChk 3.0

Syntax **int ProfInsChk( )**

This function determines if the Microsoft Windows Profiler is installed. See *Tools* for more information on using the Profiler.

This function has no parameters.

Return Value The return value specifies whether Profiler is installed and the version installed. The return value is zero if Profiler is not installed, 1 if the Windows Profiler is installed for a mode other than 386 enhanced mode, and 2 if the Windows 386 enhanced mode Profiler is installed.

ProfSampRate 3.0

Syntax **void ProfSampRate(***nRate286*, *nRate386***)**

When running the Microsoft Windows Profiler, this function sets the rate of code sampling. See *Tools* for more information on using the Profiler.

Parameter	Type/Description
nRate286	**int** Specifies the sampling rate of Profiler if the application is running with Windows in any mode other than 386 enhanced mode. The value of *nRate286* ranges from 1 to 13, indicating the following sampling rates:

Parameter	Type/Description	
	Value	**Sampling Rate**
	1	122.070 microseconds
	2	244.141 microseconds
	3	488.281 microseconds
	4	976.562 microseconds
	5	1.953125 milliseconds
	6	3.90625 milliseconds
	7	7.8125 milliseconds
	8	15.625 milliseconds
	9	31.25 milliseconds
	10	62.5 milliseconds
	11	125 milliseconds
	12	250 milliseconds
	13	500 milliseconds
nRate386	**int** Specifies the sampling rate of Profiler if the application is running with Windows in 386 enhanced mode. The value of *nRate386* can range from 1 to 1000, specifying the sampling rate in milliseconds.	

Return Value None.

Comments The default rate is 5 (1.953125 milliseconds) for Windows in any mode other than 386 enhanced mode. The default rate is 2 milliseconds for Windows in 386 enhanced mode.

Profiler only selects the parameter appropriate for the version of Windows being used.

ProfSetup 3.0

Syntax **void ProfSetup**(*nBufferSize*, *nSamples*)

When running the Microsoft Windows Profiler with Windows in 386 enhanced mode, this function specifies the size of the output buffer and the amount of samples written to disk.

Profiler ignores the **ProfSetup** function when running with Windows in any mode other than 386 enhanced mode. See *Tools* for more information on using the Profiler.

Parameter	Type/Description
nBufferSize	**int** Specifies the size of the output buffer in kilobytes. The *nBufferSize* parameter can range from 1 to 1064. The default is 64.
nSamples	**int** Specifies how much sampling data Profiler writes to disk. A value of zero specifies unlimited sampling data. The default is zero.

ProfStart 3.0

Syntax **void ProfStart( )**

When running the Microsoft Windows Profiler, this function starts sampling. See *Tools* for more information on using the Profiler.

This function has no parameters.

Return Value None.

ProfStop 3.0

Syntax **void ProfStop( )**

When running the Microsoft Windows Profiler, this function stops sampling. See *Tools* for more information on using the Profiler.

This function has no parameters.

Return Value None.

PtInRect

Syntax **BOOL PtInRect**(*lpRect*, *Point*)

This function specifies whether the specified point lies within a given rectangle. A point is within a rectangle if it lies on the left or top side, or is within all four sides. A point on the right or bottom side is outside the rectangle.

Parameter	Type/Description
lpRect	**LPRECT** Points to a **RECT** data structure that contains the specified rectangle.
Point	**POINT** Specifies a **POINT** data structure that contains the specified point.

Return Value The return value specifies whether the specified point lies within the given rectangle. It is nonzero if the point lies within the given rectangle. Otherwise, it is zero.

PtInRegion

Syntax BOOL **PtInRegion**(*hRgn*, *X*, *Y*)

This function specifies whether the point given by the *X* and *Y* parameters is in the given region.

Parameter	Type/Description
hRgn	**HRGN** Identifies the region to be examined.
X	**int** Specifies the logical *x*-coordinate of the point.
Y	**int** Specifies the logical *y*-coordinate of the point.

Return Value The return value specifies whether the specified point is in the given region. It is nonzero if the point is in the region. Otherwise, it is zero.

PtVisible

Syntax BOOL **PtVisible**(*hDC*, *X*, *Y*)

This function specifies whether the given point is within the clipping region of the specified device context.

Parameter	Type/Description
hDC	**HDC** Identifies the device context.
X	**int** Specifies the logical *x*-coordinate of the point.
Y	**int** Specifies the logical *y*-coordinate of the point.

Return Value

The return value specifies whether the specified point is within the clipping region of the given display context. It is nonzero if the point is within the clipping region. Otherwise, it is zero.

ReadComm

Syntax

int **ReadComm**(*nCid*, *lpBuf*, *nSize*)

This function reads the number of characters specified by the *nSize* parameter from the communication device specified by the *nCid* parameter and copies the characters into the buffer pointed to by the *lpBuf* parameter.

Parameter	Type/Description
nCid	**int** Specifies the communication device to be read. The **OpenComm** function returns this value.
lpBuf	**LPSTR** Points to the buffer that is to receive the characters read.
nSize	**int** Specifies the number of characters to be read.

Return Value

The return value specifies the number of characters actually read. It is less than the number specified by *nSize* only if the number of characters in the receive queue is less than that specified by *nSize*. If it is equal to *nSize*, additional characters may be queued for the device. If the return value is zero, no characters are present.

When an error occurs, the return value is set to a value less than zero, with the absolute value being the actual number of characters read. The cause of the error can be determined by using the **GetCommError** function to retrieve the error code and status. Since errors can occur when no bytes are present, if the return value is zero, the **GetCommError** function should be used to ensure that no error occurred.

For parallel I/O ports, the return value will always be zero.

RealizePalette 3.0

Syntax

int **RealizePalette**(*hDC*)

This function maps to the system palette entries in the logical palette currently selected into a device context.

A logical color palette acts as a buffer between color-intensive applications and the system, allowing an application to use as many colors as needed without interfering with its own color display, or with colors displayed by other windows. When a window has input focus and calls **RealizePalette**, Windows ensures that it will display all the colors it requests, up to the maximum number simultaneously available on the display, and displays additional colors by matching them to available colors. In addition, Windows matches the colors requested by inactive windows that call **RealizePalette** as closely as possible to the available colors. This significantly reduces undesirable changes in the colors displayed in inactive windows.

Parameter	Type/Description
hDC	**HDC** Identifies the device context.

Return Value The return value specifies how many entries in the logical palette were mapped to different entries in the system palette. This represents the number of entries which this function remapped to accommodate changes in the system palette since the logical palette was last realized.

Rectangle

Syntax **BOOL Rectangle**(*hDC*, *X1*, *Y1*, *X2*, *Y2*)

This function draws a rectangle. The interior of the rectangle is filled by using the selected brush, and a border is drawn with the selected pen.

Parameter	Type/Description
hDC	**HDC** Identifies the device context.
X1	**int** Specifies the logical *x*-coordinate of the upper-left corner of the rectangle.
Y1	**int** Specifies the logical *y*-coordinate of the upper-left corner of the rectangle.
X2	**int** Specifies the logical *x*-coordinate of the lower-right corner of the rectangle.
Y2	**int** Specifies the logical *y*-coordinate of the lower-right corner of the rectangle.

Return Value The return value specifies whether the rectangle is drawn. It is nonzero if the rectangle is drawn. Otherwise, it is zero.

Comments The width of the rectangle specified by the *X1*, *Y1*, *X2*, and *Y2* parameters must not exceed 32,767 units. This limit applies to the height of the rectangle as well.

The current position is neither used nor updated by this function.

RectInRegion 3.0

Syntax **BOOL RectInRegion**(*hRegion*, *lpRect*)

This function determines whether any part of the rectangle specified by the *lpRect* parameter is within the boundaries of the region identified by the *hRegion* parameter.

Parameter	Type/Description
hRegion	**HRGN** Identifies the region.
lpRect	**LPRECT** Identifies the rectangle.

Return Value The return value is TRUE if any part of the specified rectangle lies within the boundaries of the region. Otherwise, the return value is FALSE.

RectVisible

Syntax **BOOL RectVisible**(*hDC*, *lpRect*)

This function determines whether any part of the given rectangle lies within the clipping region of the specified display context.

Parameter	Type/Description
hDC	**HDC** Identifies the device context.
lpRect	**LPRECT** Points to a **RECT** data structure that contains the logical coordinates of the specified rectangle.

Return Value The return value specifies whether the rectangle is within the clipping region. It is nonzero if some portion of the given rectangle lies within the clipping region. Otherwise, it is zero.

RegisterClass

Syntax **BOOL RegisterClass**(*lpWndClass*)

This function registers a window class for subsequent use in calls to the **CreateWindow** function. The window class has the attributes defined by the contents of the data structure pointed to by the *lpWndClass* parameter. If two classes with the same name are registered, the second attempt fails and the information for that class is ignored.

Parameter	Type/Description
lpWndClass	**LPWNDCLASS** Points to a **WNDCLASS** data structure. The structure must be filled with the appropriate class attributes before being passed to the function. See the following "Comments" section for details.

Return Value The return value specifies whether the window class is registered. It is nonzero if the class is registered. Otherwise, it is zero.

Comments The callback function must use the Pascal calling conventions and must be declared **FAR.**

Callback Function

BOOL FAR PASCAL *WndProc*(*hWnd*, *wMsg*, *wParam*, *lParam*)
HWND *hWnd*;
WORD *wMsg*;
WORD *wParam*;
DWORD *lParam*;

WndProc is a placeholder for the application-supplied function name. The actual name must be exported by including it in an **EXPORTS** statement in the application's module-definition file.

Parameter	Description
hWnd	Identifies the window that receives the message.
wMsg	Specifies the message number.
wParam	Specifies additional message-dependent information.
lParam	Specifies additional message-dependent information.

Return Value

The window function returns the result of the message processing. The possible return values depend on the actual message sent.

M – R

RegisterClipboardFormat

Syntax **WORD RegisterClipboardFormat**(*lpFormatName*)

This function registers a new clipboard format whose name is pointed to by the *lpFormatName* parameter. The registered format can be used in subsequent clipboard functions as a valid format in which to render data, and it will appear in the clipboard's list of formats.

Parameter	Type/Description
lpFormatName	**LPSTR** Points to a character string that names the new format. The string must be a null-terminated character string.

Return Value The return value specifies the newly registered format. If the identical format name has been registered before, even by a different application, the format's reference count is increased and the same value is returned as when the format was originally registered. The return value is zero if the format cannot be registered.

Comments The format value returned by the **RegisterClipboardFormat** function is within the range of 0xC000 to 0xFFFF.

RegisterWindowMessage

Syntax **WORD RegisterWindowMessage**(*lpString*)

This function defines a new window message that is guaranteed to be unique throughout the system. The returned message value can be used when calling the **SendMessage** or **PostMessage** function.

RegisterWindowMessage is typically used for communication between two cooperating applications.

If the same message string is registered by two different applications, the same message value is returned. The message remains registered until the user ends the Windows session.

Parameter	Type/Description
lpString	**LPSTR** Points to the message string to be registered.

Return Value The return value specifies the outcome of the function. It is an unsigned short integer within the range 0xC000 to 0xFFFF if the message is successfully registered. Otherwise, it is zero.

Comments Use the **RegisterWindowMessage** function only when the same message must be understood by more than one application. For sending private messages within an application, an application can use any integer within the range WM_USER to 0xBFFF.

ReleaseCapture

Syntax **void ReleaseCapture( )**

This function releases the mouse capture and restores normal input processing. A window with the mouse capture receives all mouse input regardless of the position of the cursor.

This function has no parameters.

Return Value None.

Comments An application calls this function after calling the **SetCapture** function.

ReleaseDC

Syntax **int ReleaseDC**(*hWnd*, *hDC*)

This function releases a device context, freeing it for use by other applications. The effect of the **ReleaseDC** function depends on the device-context type. It only frees common and window device contexts. It has no effect on class or private device contexts.

Parameter	Type/Description
hWnd	**HWND** Identifies the window whose device context is to be released.
hDC	**HDC** Identifies the device context to be released.

Return Value The return value specifies whether the device context is released. It is 1 if the device context is released. Otherwise, it is zero.

Comments The application must call the **ReleaseDC** function for each call to the **GetWindowDC** function and for each call to the **GetDC** function that retrieves a common device context.

RemoveFontResource

Syntax **BOOL RemoveFontResource**(*lpFilename*)

This function removes an added font resource from the file named by the *lpFilename* parameter or from the Windows font table.

Parameter	Type/Description
lpFilename	**LPSTR** Points to a string that names the font-resource file or contains a handle to a loaded module. If *lpFilename* points to the font-resource filename, the string must be null-terminated and have the DOS filename format. If *lpFilename* contains a handle, the handle must be in the low-order word; the high-order word must be zero.

Return Value The return value specifies the outcome of the function. It is nonzero if the function is successful. Otherwise, it is zero.

Comments Any application that adds or removes fonts from the Windows font table should notify other windows of the change by using the **SendMessage** function with the *hWnd* parameter set to –1 to send a WM_FONTCHANGE message to all top-level windows in the system.

The **RemoveFontResource** function may not actually remove the font resource. If there are outstanding references to the resource, the font resource remains loaded until the last referencing logical font has been deleted by using the **DeleteObject** function.

RemoveMenu 3.0

Syntax **BOOL RemoveMenu**(*hMenu*, *nPosition*, *wFlags*)

This function deletes an menu item with an associated pop-up menu from the menu identified by the *hMenu* parameter but does not destroy the handle for the pop-up menu, allowing the menu to be reused. Before calling this function, the application should call **GetSubMenu** to retrieve the pop-up menu handle.

Parameter	Type/Description
hMenu	**HMENU** Identifies the menu to be changed.
nPosition	**WORD** Specifies the menu item to be removed. The interpretation of the *nPosition* parameter depends upon the setting of the *wFlags* parameter.

Parameter	Type/Description	
	If wFlags is	*nPosition*
	MF_BYCOMMAND	Specifies the command ID of the existing menu item.
	MF_BYPOSITION	Specifies the position of the menu item. The first item in the menu is at position zero.
wFlags	**WORD** Specifies how the *nPosition* parameter is interpreted. It must be either MF_BYCOMMAND or MF_BYPOSITION.	

Return Value The return value specifies the outcome of the function. It is TRUE if the function is successful. Otherwise, it is FALSE.

Comments Whenever a menu changes (whether or not the menu resides in a window that is displayed), the application should call **DrawMenuBar**.

RemoveProp

Syntax **HANDLE RemoveProp(***hWnd*, *lpString***)**

This function removes an entry from the property list of the specified window. The character string specified by the *lpString* parameter identifies the entry to be removed.

The **RemoveProp** function returns the data handle associated with the string so that the application can free the data associated with the handle.

Parameter	Type/Description
hWnd	**HWND** Identifies the window whose property list is to be changed.
lpString	**LPSTR** Points to a null-terminated character string or to an atom that identifies a string. If an atom is given, it must have been previously created by means of the **AddAtom** function. The atom, a 16-bit value, must be placed in the low-order word of *lpString*; the high-order word must be zero.

Return Value The return value identifies the given string. It is NULL if the string cannot be found in the given property list.

Comments An application must free the data handles associated with entries removed from a property list. The application should only remove those properties which it added to the property list.

ReplyMessage

Syntax **void ReplyMessage**(*lReply*)

This function is used to reply to a message sent through the **SendMessage** function without returning control to the function that called **SendMessage.**

By calling this function, the window function that receives the message allows the task that called **SendMessage** to continue to execute as though the task that received the message had returned control. The task that calls **ReplyMessage** also continues to execute.

Normally a task that calls **SendMessage** to send a message to another task will not continue executing until the window procedure that Windows calls to receive the message returns. However, if a task that is called to receive a message needs to perform some type of operation that might yield control (such as calling the **MessageBox** or **DialogBox** functions), Windows could be placed in a deadlock situation where the sending task needs to execute and process messages but cannot because it is waiting for **SendMessage** to return. An application can avoid this problem if the task receiving the message calls **ReplyMessage** before performing any operation that could cause the task to yield.

The **ReplyMessage** function has no effect if the message was not sent through the **SendMessage** function or if the message was sent by the same task.

Parameter	Type/Description
lReply	**LONG** Specifies the result of the message processing. The possible values depend on the actual message sent.

Return Value None.

M – R

ResizePalette 3.0

Syntax **BOOL ResizePalette**(*hPalette*, *nNumEntries*)

This function changes the size of the logical palette specified by the *hPalette* parameter to the number of entries specified by the **nNumEntries** parameter. If an application calls **ResizePalette** to reduce the size of the palette, the entries remaining in the resized palette are unchanged. If the application calls **ResizePalette** to enlarge the palette, the additional palette entries are set to black (the red, green, and blue values are all 0) and the flags for all additional entries are set to 0.

Parameter	Type/Description
hPalette	**HPALETTE** Identifies the palette to be changed.
nNumEntries	**int** Specifies the number of entries in the palette after it has been resized.

Return Value The return value specifies the outcome of the function. It is TRUE if the palette was successfully resized. Otherwise, it is FALSE.

RestoreDC

Syntax **BOOL RestoreDC**(*hDC*, *nSavedDC*)

This function restores the device context specified by the *hDC* parameter to the previous state identified by the *nSavedDC* parameter. The **RestoreDC** function restores the device context by copying state information saved on the context stack by earlier calls to the **SaveDC** function.

The context stack can contain the state information for several device contexts. If the context specified by *nSavedDC* is not at the top of the stack, **RestoreDC** deletes any state information between the device context specified by the *nSavedDC* parameter and the top of the stack. The deleted information is lost.

Parameter	Type/Description
hDC	**HDC** Identifies the device context.
nSavedDC	**int** Specifies the device context to be restored. It can be a value returned by a previous **SaveDC** function call. If *nSavedDC* is –1, the most recent device context saved is restored.

Return Value The return value specifies the outcome of the function. It is TRUE if the specified context was restored. Otherwise, it is FALSE.

RGB

Syntax **COLORREF RGB**(*cRed*, *cGreen*, *cBlue*)

This macro selects an RGB color based on the parameters supplied and the color capabilities of the output device.

Parameter	Type/Description
cRed	**BYTE** Specifies the intensity of the red color field.
cGreen	**BYTE** Specifies the intensity of the green color field.
cBlue	**BYTE** Specifies the intensity of the blue color field.

Return Value The return value specifies the resultant RGB color.

Comments The intensity for each argument can range from 0 to 255. If all three intensities are specified as 0, the result is black. If all three intensities are specified as 255, the result is white.

For information on using color values in a color palette, see the descriptions of the **PALETTEINDEX** and **PALETTERGB** macros, earlier in this chapter.

RoundRect

Syntax **BOOL RoundRect**(*hDC*, *X1*, *Y1*, *X2*, *Y2*, *X3*, *Y3*)

This function draws a rectangle with rounded corners. The interior of the rectangle is filled by using the selected brush, and a border is drawn with the selected pen.

Parameter	Type/Description
hDC	**HDC** Identifies the device context.
X1	**int** Specifies the logical *x*-coordinate of the upper-left corner of the rectangle.
Y1	**int** Specifies the logical *y*-coordinate of the upper-left corner of the rectangle.
X2	**int** Specifies the logical *x*-coordinate of the lower-right corner of the rectangle.
Y2	**int** Specifies the logical *y*-coordinate of the lower-right corner of the rectangle.
X3	**int** Specifies the width of the ellipse used to draw the rounded corners.
Y3	**int** Specifies the height of the ellipse used to draw the rounded corners.

Return Value The return value specifies whether the rectangle is drawn. It is nonzero if the rectangle is drawn. Otherwise, it is zero.

Comments The width of the rectangle specified by the *X1*, *Y1*, *X2*, and *Y2* parameters must not exceed 32,767 units. This limit applies to the height of the rectangle as well.

The current position is neither used nor updated by this function.

SaveDC

Syntax **int** SaveDC(*hDC*)

This function saves the current state of the device context specified by the *hDC* parameter by copying state information (such as clipping region, selected objects, and mapping mode) to a context stack. The saved device context can later be restored by using the **RestoreDC** function.

Parameter	Type/Description
hDC	**HDC** Identifies the device context to be saved.

Return Value The return value specifies the saved device context. It is zero if an error occurs.

Comments The **SaveDC** function can be used any number of times to save any number of device-context states.

ScaleViewportExt

Syntax **DWORD** **ScaleViewportExt**(*hDC*, *Xnum*, *Xdenom*, *Ynum*, *Ydenom*)

This function modifies the viewport extents relative to the current values. The formulas are written as follows:

xNewVE = (xOldVE × Xnum)/ X denom
yNewVE = (yOldVE × Ynum) / Ydenom

The new extent is calculated by multiplying the current extents by the given numerator and then dividing by the given denominator.

Parameter	Type/Description
hDC	**HDC** Identifies the device context.
Xnum	**int** Specifies the amount by which to multiply the current *x*-extent.
Xdenom	**int** Specifies the amount by which to divide the current *x*-extent.
Ynum	**int** Specifies the amount by which to multiply the current *y*-extent.
Ydenom	**int** Specifies the amount by which to divide the current *y*-extent.

Return Value The return value specifies the previous viewport extents (in device units). The previous *y*-extent is in the high-order word; the previous *x*-extent is in the low-order word.

ScaleWindowExt

Syntax **DWORD ScaleWindowExt**(*hDC*, *Xnum*, *Xdenom*, *Ynum*, *Ydenom*)

This function modifies the window extents relative to the current values. The formulas are written as follows:

xNewWE = (xOldWE × Xnum) / Xdenom
yNewWE = (yOldWE × Ynum) / Ydenom

The new extent is calculated by multiplying the current extents by the given numerator and then dividing by the given denominator.

Parameter	Type/Description
hDC	**HDC** Identifies the device context.
Xnum	**int** Specifies the amount by which to multiply the current *x*-extent.
Xdenom	**int** Specifies the amount by which to divide the current *x*-extent.
Ynum	**int** Specifies the amount by which to multiply the current *y*-extent.
Ydenom	**int** Specifies the amount by which to divide the current *y*-extent.

Return Value The return value specifies the previous window extents (in logical units). The previous *y*-extent is in the high-order word; the previous *x*-extent is in the low-order word.

ScreenToClient

Syntax **void ScreenToClient**(*hWnd*, *lpPoint*)

This function converts the screen coordinates of a given point on the display to client coordinates. The **ScreenToClient** function uses the window given by the *hWnd* parameter and the screen coordinates given in the **POINT** data structure pointed to by the *lpPoint* parameter to compute client coordinates, and then replaces the screen coordinates with the client coordinates. The new coordinates are relative to the upper-left corner of the given window's client area.

Parameter	Type/Description
hWnd	**HWND** Identifies the window whose client area will be used for the conversion.
lpPoint	**LPPOINT** Points to a **POINT** data structure that contains the screen coordinates to be converted.

Return Value None.

Comments The **ScreenToClient** formula assumes the given point is in screen coordinates.

ScrollDC

Syntax **BOOL ScrollDC**(*hDC*, *dx*, *dy*, *lprcScroll*, *lprcClip*, *hrgnUpdate*, *lprcUpdate*)

This function scrolls a rectangle of bits horizontally and vertically. The *lprcScroll* parameter points to the rectangle to be scrolled, the *dx* parameter specifies the number of units to be scrolled horizontally, and the *dy* parameter specifies the number of units to be scrolled vertically.

Parameter	Type/Description
hDC	**HDC** Identifies the device context that contains the bits to be scrolled.
dx	**int** Specifies the number of horizontal scroll units.
dy	**int** Specifies the number of vertical scroll units.
lprcScroll	**LPRECT** Points to the **RECT** data structure that contains the coordinates of the scrolling rectangle.
lprcClip	**LPRECT** Points to the **RECT** data structure that contains the coordinates of the clipping rectangle. When this rectangle is smaller than the original pointed to by *lprcScroll*, scrolling occurs only in the smaller rectangle.
hrgnUpdate	**HRGN** Identifies the region uncovered by the scrolling process. The **ScrollDC** function defines this region; it is not necessarily a rectangle.
lprcUpdate	**LPRECT** Points to the **RECT** data structure that, upon return, contains the coordinates of the rectangle that bounds the scrolling update region. This is the largest rectangular area that requires repainting.

Return Value This value specifies the outcome of the function. It is nonzero if scrolling is executed. Otherwise, it is zero.

Comments If the *lprcUpdate* parameter is NULL, Windows does not compute the update rectangle. If both the *hrgnUpdate* and *lprcUpdate* parameters are NULL, Windows does not compute the update region. If *hrgnUpdate* is not NULL, Windows assumes that it contains a valid region handle to the region uncovered by the scrolling process (defined by the **ScrollDC** function).

An application should use the **ScrollWindow** function when it is necessary to scroll the entire client area of a window. Otherwise, it should use **ScrollDC**.

ScrollWindow

Syntax

void ScrollWindow(*hWnd*, *XAmount*, *YAmount*, *lpRect*, *lpClipRect*)

This function scrolls a window by moving the contents of the window's client area the number of units specified by the *XAmount* parameter along the screen's *x*-axis and the number of units specified by the *YAmount* parameter along the *y*-axis. The scroll moves right if *XAmount* is positive and left if it is negative. The scroll moves down if *YAmount* is positive and up if it is negative.

Parameter	Type/Description
hWnd	**HWND** Identifies the window whose client area is to be scrolled.
XAmount	**int** Specifies the amount (in device units) to scroll in the *x* direction.
YAmount	**int** Specifies the amount (in device units) to scroll in the *y* direction.
lpRect	**LPRECT** Points to a **RECT** data structure that specifies the portion of the client area to be scrolled. If *lpRect* is NULL, the entire client area is scrolled.
lpClipRect	**LPRECT** Points to a **RECT** data structure that specifies the clipping rectangle to be scrolled. Only bits inside this rectangle are scrolled. If *lpClipRect* is NULL, the entire window is scrolled.

Return Value

None.

Comments

If the caret is in the window being scrolled, **ScrollWindow** automatically hides the caret to prevent it from being erased, then restores the caret after the scroll is finished. The caret position is adjusted accordingly.

The area uncovered by the **ScrollWindow** function is not repainted, but is combined into the window's update region. The application will eventually receive a WM_PAINT message notifying it that the region needs repainting. To repaint the uncovered area at the same time the scrolling is done, call the **UpdateWindow** function immediately after calling **ScrollWindow**.

If the *lpRect* parameter is NULL, the positions of any child windows in the window are offset by the amount specified by *XAmount* and *YAmount*, and any invalid (unpainted) areas in the window are also offset. **ScrollWindow** is faster when *lpRect* is NULL.

If the *lpRect* parameter is not NULL, the positions of child windows are *not* changed, and invalid areas in the window are *not* offset. To prevent updating problems when *lpRect* is not NULL, call the **UpdateWindow** function to repaint the window before calling **ScrollWindow**.

SelectClipRgn

Syntax

int SelectClipRgn(*hDC*, *hRgn*)

This function selects the given region as the current clipping region for the specified device context. Only a copy of the selected region is used. The region itself can be selected for any number of other device contexts, or it can be deleted.

Parameter	Type/Description
hDC	**HDC** Identifies the device context.
hRgn	**HRGN** Identifies the region to be selected.

Return Value

The return value specifies the region's type. It can be any one of the following values:

Value	Meaning
COMPLEXREGION	New clipping region has overlapping borders.
ERROR	Device context or region handle is not valid.
NULLREGION	New clipping region is empty.
SIMPLEREGION	New clipping region has no overlapping borders.

Comments

The **SelectClipRgn** function assumes that the coordinates for the given region are specified in device units.

Some printer devices support graphics at lower resolutions than text output to increase speed, but at the expense of quality. These devices scale coordinates for graphics so that one graphics device point corresponds to two or four true device points. This scaling factor affects clipping. If a region will be used to clip graphics, its coordinates must be divided down by the scaling factor. If the region will be used to clip text, no scaling adjustment is needed. The scaling factor is determined by using the **GETSCALINGFACTOR** printer escape.

SelectObject

Syntax **HANDLE SelectObject**(*hDC*, *hObject*)

This function selects the logical object specified by the *hObject* parameter as the selected object of the specified device context. The new object replaces the previous object of the same type. For example, if *hObject* is the handle to a logical pen, the **SelectObject** function replaces the selected pen with the pen specified by *hObject*.

Selected objects are the default objects used by the GDI output functions to draw lines, fill interiors, write text, and clip output to specific areas of the device surface. Although a device context can have six selected objects (pen, brush, font, bitmap, region, and logical palette), no more than one object of any given type can be selected at one time. **SelectObject** does not select a logical palette; to select a logical palette, the application must use **SelectPalette**.

Parameter	Type/Description
hDC	**HDC** Identifies the device context.
hObject	**HANDLE** Identifies the object to be selected. It may be any one of the following, and must have been created by using one of the following functions:

Object	Function
Bitmap (Bitmaps can be selected for memory device contexts only, and for only one device context at a time.)	**CreateBitmap** **CreateBitmapIndirect** **CreateCompatibleBitmap** **CreateDIBitmap**
Brush	**CreateBrushIndirect** **CreateHatchBrush** **CreatePatternBrush** **CreateSolidBrush**
Font	**CreateFont** **CreateFontIndirect**
Pen	**CreatePen** **CreatePenIndirect**
Region	**CombineRgn** **CreateEllipticRgn** **CreateEllipticRgnIndirect** **CreatePolygonRgn** **CreateRectRgn** **CreateRectRgnIndirect**

Return Value

The return value identifies the object being replaced by the object specified by the *hObject* parameter. It is NULL if there is an error.

If the *hDC* parameter specifies a metafile, the return value is nonzero if the function is successful. Otherwise, it is zero.

If a region is being selected, the return is the same as for **SelectClipRgn**.

Comments

When you select a font, pen, or brush by using the **SelectObject** function, GDI allocates space for that object in its data segment. Because data-segment space is limited, you should use the **DeleteObject** function to delete each drawing object that you no longer need.

Before deleting the last of the unneeded drawing objects, an application should select the original (default) object back into the device context.

An application cannot select a bitmap into more than one device context at any time.

SelectPalette 3.0

Syntax

HPALETTE **SelectPalette**(*hDC*, *hPalette*, *bForceBackground*)

This function selects the logical palette specified by the *hPalette* parameter as the selected palette object of the device context identified by the *hDC* parameter. The new palette becomes the palette object used by GDI to control colors displayed in the device context and replaces the previous palette.

Parameter	Type/Description
hDC	**HDC** Identifies the device context.
hPalette	**HPALETTE** Identifies the logical palette to be selected. **CreatePalette** creates a logical palette.
bForceBackground	**BOOL** Specifies whether the logical palette is forced to be a background palette. If *bForceBackground* is nonzero, the selected palette is always a background palette, regardless of whether the window has input focus. If *bForceBackground* is zero, the logical palette is a foreground palette when the window has input focus.

Return Value

The return value identifies the logical palette being replaced by the palette specified by the *hPalette* parameter. It is NULL if there is an error.

Comments An application can select a logical palette into more than one device context. However, changes to a logical palette will affect all device contexts for which it is selected. If an application selects a palette object into more than one device context, the device contexts must all belong to the same physical device (such as a display or printer).

SendDlgItemMessage

Syntax DWORD **SendDlgItemMessage**(*hDlg*, *nIDDlgItem*, *wMsg*, *wParam*, *lParam*)

This function sends a message to the control specified by the *nIDDlgItem* parameter within the dialog box specified by the *hDlg* parameter. The **SendDlgItemMessage** function does not return until the message has been processed.

Parameter	Type/Description
hDlg	**HWND** Identifies the dialog box that contains the control.
nIDDlgItem	**int** Specifies the integer identifier of the dialog item that is to receive the message.
wMsg	**WORD** Specifies the message value.
wParam	**WORD** Specifies additional message information.
lParam	**DWORD** Specifies additional message information.

Return Value The return value specifies the outcome of the function. It is the value returned by the control's window function, or zero if the control identifier is not valid.

Comments Using **SendDlgItemMessage** is identical to obtaining a handle to the given control and calling the **SendMessage** function.

SendMessage

Syntax DWORD **SendMessage**(*hWnd*, *wMsg*, *wParam*, *lParam*)

This function sends a message to a window or windows. The **SendMessage** function does not return until the message has been processed. If the window that receives the message is part of the same application, the window function is called immediately as a subroutine. If the window is part of another task, Windows switches to the appropriate task and calls the appropriate window function, and then passes the message to the window function. The message is not placed in the destination application's queue.

Parameter	Type/Description
hWnd	**HWND** Identifies the window that is to receive the message. If the *hWnd* parameter is 0xFFFF, the message is sent to all pop-up windows in the system. The message is not sent to child windows.
wMsg	**WORD** Specifies the message to be sent.
wParam	**WORD** Specifies additional message information.
lParam	**DWORD** Specifies additional message information.

Return Value The return value specifies the outcome of the function. It is the value returned by the window function that received the message; its value depends on the message being sent.

Comments If a system running Windows is configured for expanded memory (EMS) and an application sends a message (by using the **SendMessage** function) with related data (that is pointed to by the *lParam* parameter) to a second application, the first application must place the data (that *lParam* points to) in global memory allocated by the **GlobalAlloc** function and the GMEM_LOWER flag. Note that this allocation of memory is only necessary if *lParam* contains a pointer.

SetActiveWindow

Syntax **HWND SetActiveWindow**(*hWnd*)

This function makes a top-level window the active window.

Parameter	Type/Description
hWnd	**HWND** Identifies the top-level window to be activated.

Return Value The return value identifies the window that was previously active. The **SetActiveWindow** function should be used with care since it allows an application to arbitrarily take over the active window and input focus. Normally, Windows takes care of all activation.

SetBitmapBits

Syntax **LONG SetBitmapBits**(*hBitmap*, *dwCount*, *lpBits*)

This function sets the bits of a bitmap to the bit values given by the *lpBits* parameter.

Parameter	Type/Description
hBitmap	**HBITMAP** Identifies the bitmap to be set.
dwCount	**DWORD** Specifies the number of bytes pointed to by *lpBits*.
lpBits	**LPSTR** Points to the bitmap bits that are stored as a long pointer to a byte array.

Return Value The return value specifies the number of bytes used in setting the bitmap bits. It is zero if the function fails.

SetBitmapDimension

Syntax **DWORD SetBitmapDimension**(*hBitmap*, *X*, *Y*)

This function assigns a width and height to a bitmap in 0.1-millimeter units. These values are not used internally by GDI; the **GetBitmapDimension** function can be used to retrieve them.

Parameter	Type/Description
hBitmap	**HANDLE** Identifies the bitmap.
X	**int** Specifies the width of the bitmap (in 0.1-millimeter units).
Y	**int** Specifies the height of the bitmap (in 0.1-millimeter units).

Return Value The return value specifies the previous bitmap dimensions. Height is in the high-order word, and width is in the low-order word.

SetBkColor

Syntax **DWORD SetBkColor**(*hDC*, *crColor*)

This function sets the current background color to the color specified by the *crColor* parameter, or to the nearest physical color if the device cannot represent an RGB color value specified by *crColor*.

If the background mode is OPAQUE, GDI uses the background color to fill the gaps between styled lines, gaps between hatched lines in brushes, and character cells. GDI also uses the background color when converting bitmaps from color to monochrome and vice versa.

The background mode is set by the **SetBkMode** function. See the **BitBlt** and **StretchBlt** functions, in this chapter, for color-bitmap conversions.

Parameter	Type/Description
hDC	**HDC** Identifies the device context.
crColor	**COLORREF** Specifies the new background color.

Return Value The return value specifies the previous background color as an RGB color value. If an error occurs, the return value is 0x80000000.

SetBkMode

Syntax **int SetBkMode**(*hDC*, *nBkMode*)

This function sets the background mode used with text and line styles. The background mode defines whether or not GDI should remove existing background colors on the device surface before drawing text, hatched brushes, or any pen style that is not a solid line.

Parameter	Type/Description
hDC	**HDC** Identifies the device context.
nBkMode	**int** Specifies the background mode. It can be either one of the following modes:

Value	Meaning
OPAQUE	Background is filled with the current background color before the text, hatched brush, or pen is drawn.
TRANSPARENT	Background remains untouched.

Return Value The return value specifies the previous background mode. It can be either OPAQUE or TRANSPARENT.

SetBrushOrg

Syntax **DWORD SetBrushOrg**(*hDC*, *X*, *Y*)

This function sets the origin of the brush currently selected into the given device context.

Parameter	Type/Description
hDC	**HDC** Identifies the device context.
X	**int** Specifies the *x*-coordinate (in device units) of the new origin. This value must be in the range 0–7.
Y	**int** Specifies the *y*-coordinate (in device units) of the new origin. This value must be in the range 0–7.

Return Value The return value specifies the origin of the brush. The previous *x*-coordinate is in the low-order word, and the previous *y*-coordinate is in the high-order word.

Comments The original brush origin is at the coordinate (0,0).

The **SetBrushOrg** function should not be used with stock objects.

SetCapture

Syntax **HWND SetCapture**(*hWnd*)

This function causes all subsequent mouse input to be sent to the window specified by the *hWnd* parameter, regardless of the position of the cursor.

Parameter	Type/Description
hWnd	**HWND** Identifies the window that is to receive the mouse input.

Return Value The return value identifies the window that previously received all mouse input. It is NULL if there is no such window.

Comments When the window no longer requires all mouse input, the application should call the **ReleaseCapture** function so that other windows can receive mouse input.

SetCaretBlinkTime

Syntax **void SetCaretBlinkTime**(*wMSeconds*)

This function sets the caret blink rate (elapsed time between caret flashes) to the number of milliseconds specified by the *wMSeconds* parameter. The caret flashes on or off each *wMSeconds* milliseconds. This means one complete flash (on-off-on) takes 2 × *wMSeconds* milliseconds.

Parameter	Type/Description
wMSeconds	**WORD** Specifies the new blink rate (in milliseconds).

Return Value None.

Comments The caret is a shared resource. A window should set the caret blink rate only if it owns the caret. It should restore the previous rate before it loses the input focus or becomes inactive.

SetCaretPos

Syntax **void SetCaretPos**(*X*, *Y*)

This function moves the caret to the position given by logical coordinates specified by the *X* and *Y* parameters. Logical coordinates are relative to the client area of the window that owns them and are affected by the window's mapping mode, so the exact position in pixels depends on this mapping mode.

The **SetCaretPos** function moves the caret only if it is owned by a window in the current task. **SetCaretPos** moves the caret whether or not the caret is hidden.

Parameter	Type/Description
X	**int** Specifies the new *x*-coordinate (in logical coordinates) of the caret.
Y	**int** Specifies the new *y*-coordinate (in logical coordinates) of the caret.

Return Value None.

Comments The caret is a shared resource. A window should not move the caret if it does not own the caret.

SetClassLong

Syntax **LONG SetClassLong**(*hWnd*, *nIndex*, *dwNewLong*)

This function replaces the long value specified by the *nIndex* parameter in the **WND-CLASS** data structure of the window specified by the *hWnd* parameter.

Parameter	Type/Description
hWnd	**HWND** Identifies the window.
nIndex	**int** Specifies the byte offset of the word to be changed. It can also be one of the following values:

Value	Meaning
GCL_MENUNAME	Sets a new long pointer to the menu name.
GCL_WNDPROC	Sets a new long pointer to the window function.

Parameter	Type/Description
dwNewLong	**DWORD** Specifies the replacement value.

Return Value The return value specifies the previous value of the specified long integer.

Comments If the **SetClassLong** function and GCL_WNDPROC index are used to set a window function, the given function must have the window-function form and be exported in the module-definition file. See the **RegisterClass** function earlier in this chapter for details.

Calling **SetClassLong** with the GCL_WNDPROC index creates a subclass of the window class that affects all windows subsequently created with the class. See Chapter 1, "Window Manager Interface Functions," for more information on window subclassing. An application should not attempt to create a window subclass for standard Windows controls such as combo boxes and buttons.

To access any extra two-byte values allocated when the window-class structure was created, use a positive byte offset as the index specified by the *nIndex* parameter, starting at zero for the first two-byte value in the extra space, 2 for the next two-byte value and so on.

SetClassWord

Syntax **WORD SetClassWord**(*hWnd, nIndex, wNewWord*)

This function replaces the word specified by the *nIndex* parameter in the **WNDCLASS** structure of the window specified by the *hWnd* parameter.

Parameter	Type/Description
hWnd	**HWND** Identifies the window.
nIndex	**int** Specifies the byte offset of the word to be changed. It can also be one of the following values:

Value	Meaning
GCW_CBCLSEXTRA	Sets two new bytes of additional window-class data.
GCW_CBWNDEXTRA	Sets two new bytes of additional window-class data.
GCW_HBRBACKGROUND	Sets a new handle to a background brush.
GCW_HCURSOR	Sets a new handle to a cursor.
GCW_HICON	Sets a new handle to an icon.
GCW_STYLE	Sets a new style bit for the window class.

Parameter	Type/Description
wNewWord	**WORD** Specifies the replacement value.

Return Value

The return value specifies the previous value of the specified word.

Comments

The **SetClassWord** function should be used with care. For example, it is possible to change the background color for a class by using **SetClassWord**, but this change does not cause all windows belonging to the class to be repainted immediately.

To access any extra four-byte values allocated when the window-class structure was created, use a positive byte offset as the index specified by the *nIndex* parameter, starting at zero for the first four-byte value in the extra space, 4 for the next four-byte value and so on.

SetClipboardData

Syntax

HANDLE SetClipboardData(*wFormat*, *hMem*)

This function sets a data handle to the clipboard for the data specified by the *hMem* parameter. The data are assumed to have the format specified by the *wFormat* parameter. After setting a clipboard data handle, the **SetClipboardData** function frees the block of memory identified by *hMem*.

Parameter	Type/Description
wFormat	**WORD** Specifies a data format. It can be any one of the predefined formats given in Table 4.13, "Predefined Data Formats." In addition to the predefined formats, any format value registered through the **RegisterClipboardFormat** function can be used as the *wFormat* parameter.
hMem	**HANDLE** Identifies the global memory block that contains the data in the specified format. The *hMem* parameter can be NULL. When *hMem* is NULL the application does not have to format the data and provide a handle to it until requested to do so through a WM_RENDERFORMAT message.

Return Value

The return value identifies the data and is assigned by the clipboard.

Comments

Once the *hMem* parameter has been passed to **SetClipboardData**, the block of data becomes the property of the clipboard. The application may read the data, but should not free the block or leave it locked.

Table 4.13 shows the various predefined data-format values for the *wFormat* parameter:

Table 4.13 Predefined Data Formats

Value	Meaning
CF_BITMAP	A handle to a bitmap (HBITMAP).
CF_DIB	A memory block containing a **BITMAPINFO** data structure followed by the bitmap bits.
CF_DIF	Software Arts' Data Interchange Format.
CF_DSPBITMAP	Bitmap display format associated with private format. The *hMem* parameter must be a handle to data that can be displayed in bitmap format in lieu of the privately formatted data.
CF_DSPMETAFILEPICT	Metafile-picture display format associated with private format. The *hMem* parameter must be a handle to data that can be displayed in metafile-picture format in lieu of the privately formatted data.
CF_DSPTEXT	Text display format associated with private format. The *hMem* parameter must be a handle to data that can be displayed in text format in lieu of the privately formatted data.
CF_METAFILEPICT	Metafile picture format as defined by the **METAFILEPICT** data structure.

Table 4.13 Predefined Data Formats (*continued*)

Value	Meaning
CF_OEMTEXT	Text format contining characters in the OEM character set. Each line ends with a carriage return/linefeed (CR-LF) combination. A null character signals the end of the data.
CF_OWNERDISPLAY	Owner display format. The clipboard owner must display and update the clipboard application window, and will receive WM_ASKCBFORMATNAME, WM_HSCROLLCLIPBOARD, WM_PAINTCLIPBOARD, WM_SIZECLIPBOARD, and WM_VSCROLLCLIPBOARD messages. The *hMem* parameter must be NULL.
CF_PALETTE	Handle to a color palette. Whenever an application places data in the clipboard that depends on or assumes a color palette, it should also place the palette in the clipboard as well. If the clipboard contains data in the CF_PALETTE (logical color palette) format, the application should assume that any other data in the clipboard is realized against that logical palette. The clipboard-viewer application (CLIPBRD.EXE) always uses as its current palette any object in CF_PALETTE format that is in the clipboard when it displays the other formats in the clipboard.
CF_PRIVATEFIRST to CF_PRIVATELAST	Range of integer values used for private formats. Data handles associated with formats in this range will not be freed automatically; any data handles must be freed by the application before the application terminates or when a WM_DESTROYCLIPBOARD message is received.
CF_SYLK	Microsoft Symbolic Link (SYLK) format.
CF_TEXT	Text format. Each line ends with a carriage return/linefeed (CR-LF) combination. A null character signals the end of the data.
CF_TIFF	Tag Image File Format.

Windows supports two formats for text, CF_TEXT and CF_OEMTEXT. CF_TEXT is the default Windows text clipboard format, while Windows uses the CF_OEMTEXT format for text in non-Windows applications. If you call **GetClipboardData** to retrieve data in one text format and the other text format is the only available text format, Windows automatically converts the text to the requested format before supplying it to your application.

An application registers other standard formats, such as Rich Text Format (RTF), by name using the **RegisterClipboardFormat** function rather than by a symbolic constant. For information on these external formats, see the README.TXT file.

SetClipboardViewer

Syntax

HWND **SetClipboardViewer**(*hWnd*)

This function adds the window specified by the *hWnd* parameter to the chain of windows that are notified (via the WM_DRAWCLIPBOARD message) whenever the contents of the clipboard are changed.

Parameter	Type/Description
hWnd	**HWND** Identifies the window to receive clipboard-viewer chain messages.

Return Value

The return value identifies the next window in the clipboard-viewer chain. This handle should be saved in static memory and used in responding to clipboard-viewer chain messages.

Comments

Windows that are part of the clipboard-viewer chain must respond to WM_CHANGECB-CHAIN, WM_DRAWCLIPBOARD, and WM_DESTROY messages.

If an application wishes to remove itself from the clipboard-viewer chain, it must call the **ChangeClipboardChain** function.

SetCommBreak

Syntax

int **SetCommBreak**(*nCid*)

This function suspends character transmission and places the transmission line in a break state until the **ClearCommBreak** function is called.

Parameter	Type/Description
nCid	**int** Specifies the communication device to be suspended. The **OpenComm** function returns this value.

Return Value

The return value specifies the result of the function. It is zero if the function is successful. It is negative if *nCid* does not specify a valid device.

SetCommEventMask

Syntax

WORD FAR * **SetCommEventMask**(*nCid*, *nEvtMask*)

This function enables and retrieves the event mask of the communication device specified by the *nCid* parameter. The bits of the *nEvtMask* parameter define which events are to be enabled. The return value points to the current state of the event mask.

Parameter	Type/Description
nCid	**int** Specifies the communication device to be enabled. The **OpenComm** function returns this value.
nEvtMask	**int** Specifies which events are to be enabled. It can be any combination of the values shown in Table 4.14, "Event Values."

Return Value

The return value points to an integer event mask. Each bit in the event mask specifies whether or not a given event has occurred. A bit is 1 if the event has occurred.

Comments

Table 4.14 lists the event values for the *nEvtMask* parameter:

Table 4.14 Event Values

Value	Meaning
EV_BREAK	Sets when a break is detected on input.
EV_CTS	Sets when the clear-to-send (CTS) signal changes state.
EV_DSR	Sets when the data-set-ready (DSR) signal changes state.
EV_ERR	Sets when a line-status error occurs. Line-status errors are CE_FRAME, CE_OVERRUN, and CE_RXPARITY.
EV_PERR	Sets when a printer error is detected on a parallel device. Errors are CE_DNS, CE_IOE, CE_LOOP, and CE_PTO.
EV_RING	Sets when a ring indicator is detected.
EV_RLSD	Sets when the receive-line-signal-detect (RLSD) signal changes state.
EV_RXCHAR	Sets when any character is received and placed in the receive queue.
EV_RXFLAG	Sets when the event character is received and placed in the receive queue. The event character is specified in the device's control block.
EV_TXEMPTY	Sets when the last character in the transmit queue is sent.

SetCommState

Syntax **int** **SetCommState**(*lpDCB*)

This function sets a communication device to the state specified by the device control block pointed to by the *lpDCB* parameter. The device to be set must be identified by the **Id** field of the control block.

This function reinitializes all hardware and controls as defined by *lpDCB*, but does not empty transmit or receive queues.

Parameter	Type/Description
lpDCB	**DCB FAR *** Points to a **DCB** data structure that contains the desired communications setting for the device.

Return Value The return value specifies the outcome of the function. It is zero if the function is successful. It is negative if an error occurs.

SetCursor

Syntax **HCURSOR** **SetCursor**(*hCursor*)

This function sets the cursor shape to the shape specified by the *hCursor* parameter. The cursor is set only if the new shape is different from the current shape. Otherwise, the function returns immediately. The **SetCursor** function is quite fast if the cursor identified by the *hCursor* parameter is the same as the current cursor.

If *hCursor* is NULL, the cursor is removed from the screen.

Parameter	Type/Description
hCursor	**HCURSOR** Identifies the cursor resource. The resource must have been loaded previously by using the **LoadCursor** function.

Return Value The return value identifies the cursor resource that defines the previous cursor shape. It is NULL if there is no previous shape.

Comments The cursor is a shared resource. A window that uses the cursor should set the shape only when the cursor is in its client area or when it is capturing all mouse input. In systems without a mouse, the window should restore the previous cursor shape before the cursor leaves the client area or before the window relinquishes control to another window.

Any application that needs to change the shape of the cursor while it is in a window must make sure the class cursor for the given window's class is set to NULL. If the class cursor is not NULL, Windows restores the previous shape each time the mouse is moved.

The cursor is not shown on the screen if the cursor display count is less than zero. This results from the **HideCursor** function being called more times than the **ShowCursor** function.

SetCursorPos

Syntax **void SetCursorPos**(*X*, *Y*)

This function moves the cursor to the screen coordinates given by the *X* and *Y* parameters. If the new coordinates are not within the screen rectangle set by the most recent **ClipCursor** function, Windows automatically adjusts the coordinates so that the cursor stays within the rectangle.

Parameter	Type/Description
X	**int** Specifies the new *x*-coordinate (in screen coordinates) of the cursor.
Y	**int** Specifies the new *y*-coordinate (in screen coordinates) of the cursor.

Return Value None.

Comments The cursor is a shared resource. A window should move the cursor only when the cursor is in its client area.

SetDIBits 3.0

Syntax **int SetDIBits**(*hDC*, *hBitmap*, *nStartScan*, *nNumScans*, *lpBits*, *lpBitsInfo*, *wUsage*)

This function sets the bits of a bitmap to the values given in a device-independent bitmap (DIB) specification.

Parameter	Type/Description
hDC	**HDC** Identifies the device context.
hBitmap	**HBITMAP** Identifies the bitmap.

Parameter	Type/Description
nStartScan	**WORD** Specifies the scan number of the first scan line in the *lpBits* buffer.
nNumScans	**WORD** Specifies the number of scan lines in the *lpBits* buffer and the number of lines to set in the bitmap identified by the *hBitmap* parameter.
lpBits	**LPSTR** Points to the device-independent bitmap bits that are stored as an array of bytes. The format of the bitmap values depends on the **biBitCount** field of the **BITMAPINFO** structure identified by *lpBitsInfo*. See the description of the **BITMAPINFO** data structure in Chapter 7, "Data Types and Structures," in *Reference, Volume 2*, for more information.
lpBitsInfo	**LPBITMAPINFO** Points to a **BITMAPINFO** data structure that contains information about the device-independent bitmap.
wUsage	**WORD** Specifies whether the **bmiColors[ ]** fields of the *lpBitsInfo* parameter contain explicit RGB values or indexes into the currently realized logical palette. The *wUsage* parameter must be one of the following values:

Value	Meaning
DIB_PAL_COLORS	The color table consists of an array of 16-bit indexes into the currently realized logical palette.
DIB_RGB_COLORS	The color table contains literal RGB values.

Return Value

The return value specifies the number of scan lines successfully copied. It is zero if the function fails.

Comments

The bitmap identified by the *hBitmap* parameter must not be selected into a device context when the application calls this function.

The origin for device-independent bitmaps is the bottom-left corner of the bitmap, not the top-left corner, which is the origin when the mapping mode is MM_TEXT.

This function also accepts a bitmap specification formatted for Microsoft OS/2 Presentation Manager versions 1.1 and 1.2 if the *lpBitsInfo* parameter points to a **BITMAPCORE-INFO** data structure.

SetDIBitsToDevice 3.0

Syntax

WORD SetDIBitsToDevice(*hDC*, *DestX*, *DestY*, *nWidth*, *nHeight*, *SrcX*, *SrcY*, *nStartScan*, *nNumScans*, *lpBits*, *lpBitsInfo*, *wUsage*)

This function sets bits from a device-independent bitmap (DIB) directly on a device surface. The *SrcX*, *SrcY*, *nWidth*, and *nHeight* parameters define a rectangle within the total DIB. **SetDIBitsToDevice** sets the bits in this rectangle directly on the display surface of the output device identified by the **hDC** parameter, at the location described by the *DestX* and *DestY* parameters.

To reduce the amount of memory required to set bits from a large DIB on a device surface, an application can band the output by repeatedly calling **SetDIBitsToDevice**, placing a different portion of the entire DIB into the *lpBits* buffer each time. The values of the *nStartScan* and *nNumScans* parameters identify the portion of the entire DIB which is contained in the *lpBits* buffer.

Parameter	Type/Description
hDC	**HDC** Identifies the device context.
DestX	**WORD** Specifies the *x*-coordinate of the origin of the destination rectangle.
DestY	**WORD** Specifies the *y*-coordinate of the origin of the destination rectangle.
nWidth	**WORD** Specifies the *x*-extent of the rectangle in the DIB.
nHeight	**WORD** Specifies the *y*-extent of the rectangle in the DIB.
SrcX	**WORD** Specifies the *x*-coordinate of the source in the DIB.
SrcY	**WORD** Specifies the *y*-coordinate of the source in the DIB.
nStartScan	**WORD** Specifies the scan-line number of the DIB which is contained in the first scan line of the *lpBits* buffer.
nNumScans	**WORD** Specifies the number of scan lines of the DIB which are contained in the *lpBits* buffer.
lpBits	**LPSTR** Points to the DIB bits that are stored as an array of bytes.
lpBitsInfo	**LPBITMAPINFO** Points to a **BITMAPINFO** data structure that contains information about the DIB.

Parameter	Type/Description	
wUsage	**WORD** Specifies whether the **bmiColors[ ]** fields of the *lpBitsInfo* parameter contain explicit RGB values or indexes into the currently realized logical palette. The *wUsage* parameter must be one of the following values:	

Value	Meaning
DIB_PAL_COLORS	The color table consists of an array of 16-bit indexes into the currently realized logical palette.
DIB_RGB_COLORS	The color table contains literal RGB values.

Return Value The return value is the number of scan lines set.

Comments All coordinates are device coordinates (that is, the coordinates of the DIB) except *destX* and *destY*, which are logical coordinates.

The origin for device-independent bitmaps is the bottom-left corner of the DIB, not the top-left corner, which is the origin when the mapping mode is MM_TEXT.

This function also accepts a device-independent bitmap specification formatted for Microsoft OS/2 Presentation Manager versions 1.1 and 1.2 if the *lpBitsInfo* parameter points to a **BITMAPCOREINFO** data structure.

SetDlgItemInt

Syntax void **SetDlgItemInt**(*hDlg*, *nIDDlgItem*, *wValue*, *bSigned*)

This function sets the text of a control in the given dialog box to the string that represents the integer value given by the *wValue* parameter. The **SetDlgItemInt** function converts *wValue* to a string that consists of decimal digits, and then copies the string to the control. If the *bSigned* parameter is nonzero, *wValue* is assumed to be signed. If *wValue* is signed and less than zero, the function places a minus sign before the first digit in the string.

SetDlgItemInt sends a WM_SETTEXT message to the given control.

Parameter	Type/Description
hDlg	**HWND** Identifies the dialog box that contains the control.
nIDDlgItem	**int** Specifies the control to be modified.

Parameter	Type/Description
wValue	**WORD** Specifies the value to be set.
bSigned	**BOOL** Specifies whether or not the integer value is signed.

Return Value None.

SetDlgItemText

Syntax **void SetDlgItemText**(*hDlg*, *nIDDlgItem*, *lpString*)

This function sets the caption or text of a control in the dialog box specified by the *hDlg* parameter. The **SetDlgItemText** function sends a WM_SETTEXT message to the given control.

Parameter	Type/Description
hDlg	**HWND** Identifies the dialog box that contains the control.
nIDDlgItem	**int** Specifies the control whose text is to be set.
lpString	**LPSTR** Points to the null-terminated character string that is to be copied to the control.

Return Value None.

SetDoubleClickTime

Syntax **void SetDoubleClickTime**(*wCount*)

This function sets the double-click time for the mouse. A double-click is a series of two clicks of the mouse button, the second occurring within a specified time after the first. The double-click time is the maximum number of milliseconds that may occur between the first and second clicks of a double-click.

Parameter	Type/Description
wCount	**WORD** Specifies the number of milliseconds that can occur between double-clicks.

Return Value None.

Comments If the *wCount* parameter is set to zero, Windows will use the default double-click time of 500 milliseconds.

The **SetDoubleClickTime** function alters the double-click time for all windows in the system.

SetEnvironment

Syntax **int SetEnvironment**(*lpPortName*, *lpEnviron*, *nCount*)

This function copies the contents of the buffer specified by the *lpEnviron* parameter into the environment associated with the device attached to the system port specified by the *lpPortName* parameter. The **SetEnvironment** function deletes any existing environment. If there is no environment for the given port, **SetEnvironment** creates one. If the *nCount* parameter is zero, the existing environment is deleted and not replaced.

Parameter	Type/Description
lpPortName	**LPSTR** Points to a null-terminated character string that specifies the name of the desired port.
lpEnviron	**LPSTR** Points to the buffer that contains the new environment.
nCount	**WORD** Specifies the number of bytes to be copied.

Return Value The return value specifies the actual number of bytes copied to the environment. It is zero if there is an error. It is –1 if the environment is deleted.

Comments The first field in the buffer pointed to by the *lpEnviron* parameter must be the same as that passed in the *lpDeviceName* parameter of the **CreateDC** function. If *lpPortName* specifies a null port (as defined in the WIN.INI file), the device name pointed to by *lpEnviron* is used to locate the desired environment.

SetErrorMode

S

Syntax **WORD SetErrorMode** (*wMode*)

This function controls whether Windows handles DOS Function 24H errors or allows the calling application to handle them.

Windows intercepts all INT 24H errors. If the application calls **SetErrorMode** with the *wMode* parameter set to zero and an INT 24H error subsequently occurs, Windows displays an error message box. If the application calls **SetErrorMode** with *wMode* set to 1 and an INT 24H occurs, Windows does not display the standard INT 24H error message box, but rather fails the original INT 21H call back to the application. This allows the application to handle disk errors using INT 21H, AH=59H (**Get Extended Error**) as appropriate.

Parameter	Type/Description
wMode	**WORD** Specifies the error mode flag. If bit 0 is set to zero, Windows displays an error message box when an INT 24H error occurs. If bit 0 is set to 1, Windows fails the INT 21H call to the calling application and does not display a message box.

Return Value The return value specifies the previous of the error mode flag.

SetFocus

Syntax **HWND** SetFocus(*hWnd*)

This function assigns the input focus to the window specified by the *hWnd* parameter. The input focus directs all subsequent keyboard input to the given window. The window, if any, that previously had the input focus loses it. If *hWnd* is NULL, key strokes are ignored.

The **SetFocus** function sends a WM_KILLFOCUS message to the window that loses the input focus and a WM_SETFOCUS message to the window that receives the input focus. It also activates either the window that receives the focus or the parent of the window that receives the focus.

Parameter	Type/Description
hWnd	**HWND** Identifies the window to receive the keyboard input.

Return Value The return value identifies the window that previously had the input focus. It is NULL if there is no such window.

Comments If a window is active but doesn't have the focus (that is, no window has the focus), any key pressed will produce the WM_SYSCHAR, WM_SYSKEYDOWN, or WM_SYSKEYUP message. If the VK_MENU key is also pressed, the *lParam* parameter of the message will have bit 30 set. Otherwise, the messages that are produced do *not* have this bit set.

SetHandleCount 3.0

Syntax **WORD SetHandleCount**(*wNumber*)

This function changes the number of file handles available to a task. By default, the maximum number of file handles available to a task is 20.

Parameter	Type/Description
wNumber	**WORD** Specifies the number of file handles needed by the application. The maximum is 255.

Return Value The return value specifies the number of file handles actually available to the application. It may be less than the number specified by the *wNumber* parameter.

SetKeyboardState

Syntax **void SetKeyboardState**(*lpKeyState*)

This function copies the 256 bytes pointed to by the *lpKeyState* parameter into the Windows keyboard-state table.

Parameter	Type/Description
lpKeyState	**BYTE FAR *** Points to an array of 256 bytes that contains keyboard key states.

Return Value None.

Comments In many cases, an application should call the **GetKeyboardState** function first to initialize the 256-byte array. The application should then change the desired bytes.

SetKeyboardState sets the LEDs and BIOS flags for the NUMLOCK, CAPSLOCK, and SCROLL LOCK keys according to the toggle state of the VK_NUMLOCK, VK_CAPITAL, and VK_OEM_SCROLL entries of the array.

For more information, see the description of **GetKeyboardState**, earlier in this chapter.

SetMapMode

Syntax **int SetMapMode**(*hDC*, *nMapMode*)

This function sets the mapping mode of the specified device context. The mapping mode defines the unit of measure used to transform logical units into device units, and also defines the orientation of the device's *x*- and *y*-axes. GDI uses the mapping mode to convert logical coordinates into the appropriate device coordinates.

Parameter	Type/Description
hDC	**HDC** Identifies the device context.
nMapMode	**int** Specifies the new mapping mode. It can be any one of the values shown in Table 4.15, "Mapping Modes."

Return Value The return value specifies the previous mapping mode.

Comments The MM_TEXT mode allows applications to work in device pixels, whose size varies from device to device.

The MM_HIENGLISH, MM_HIMETRIC, MM_LOENGLISH, MM_LOMETRIC, and MM_TWIPS modes are useful for applications that need to draw in physically meaningful units (such as inches or millimeters).

The MM_ISOTROPIC mode ensures a 1:1 aspect ratio, which is useful when preserving the exact shape of an image is important.

The MM_ANISOTROPIC mode allows the *x*- and *y*-coordinates to be adjusted independently.

Table 4.15 shows the value and meaning of the various mapping modes:

Table 4.15 Mapping Modes

Value	Meaning
MM_ANISOTROPIC	Logical units are mapped to arbitrary units with arbitrarily scaled axes. The **SetWindowExt** and **SetViewportExt** functions must be used to specify the desired units, orientation, and scaling.
MM_HIENGLISH	Each logical unit is mapped to 0.001 inch. Positive *x* is to the right; positive *y* is up.
MM_HIMETRIC	Each logical unit is mapped to 0.01 millimeter. Positive *x* is to the right; positive *y* is up.

Table 4.15 Mapping Modes (*continued*)

Value	Meaning
MM_ISOTROPIC	Logical units are mapped to arbitrary units with equally scaled axes; that is, one unit along the x-axis is equal to one unit along the y-axis. The **SetWindowExt** and **SetViewportExt** functions must be used to specify the desired units and the orientation of the axes. GDI makes adjustments as necessary to ensure that the x and y units remain the same size.
MM_LOENGLISH	Each logical unit is mapped to 0.01 inch. Positive x is to the right; positive y is up.
MM_LOMETRIC	Each logical unit is mapped to 0.1 millimeter. Positive x is to the the right; positive y is up.
MM_TEXT	Each logical unit is mapped to one device pixel. Positive x is to the right; positive y is down.
MM_TWIPS	Each logical unit is mapped to one twentieth of a printer's point (1/1440 inch). Positive x is to the right; positive y is up.

SetMapperFlags

Syntax **DWORD SetMapperFlags**(*hDC*, *dwFlag*)

This function alters the algorithm that the font mapper uses when it maps logical fonts to physical fonts. When the first bit of the *wFlag* parameter is set to 1, the mapper will only select fonts whose x-aspect and y-aspect exactly match those of the specified device. If no fonts exist with a matching aspect height and width, GDI chooses an aspect height and width and selects fonts with aspect heights and widths that match the one chosen by GDI.

Parameter	Type/Description
hDC	**HDC** Identifies the device context that contains the font-mapper flag.
dwFlag	**DWORD** Specifies whether the font mapper attempts to match a font's aspect height and width to the device. When the first bit is set to 1, the mapper will only select fonts whose x-aspect and y-aspect exactly match those of the specified device.

Return Value The return value specifies the previous value of the font-mapper flag.

Comments The remaining bits of the *dwFlag* parameter must be zero.

SetMenu

Syntax **BOOL SetMenu**(*hWnd*, *hMenu*)

This function sets the given window's menu to the menu specified by the *hMenu* parameter. If *hMenu* is NULL, the window's current menu is removed. The **SetMenu** function causes the window to be redrawn to reflect the menu change.

Parameter	Type/Description
hWnd	**HWND** Identifies the window whose menu is to be changed.
hMenu	**HMENU** Identifies the new menu.

Return Value The return value specifies whether the menu is changed. It is nonzero if the menu is changed. Otherwise, it is zero.

Comments **SetMenu** will not destroy a previous menu. An application should call the **DestroyMenu** function to accomplish this task.

SetMenuItemBitmaps 3.0

Syntax **BOOL SetMenuItemBitmaps**(*hMenu*, *nPosition*, *wFlags*, *hBitmapUnchecked*, *hBitmapChecked*)

This function associates the specified bitmaps with a menu item. Whether the menu item is checked or unchecked, Windows displays the appropriate bitmap next to the menu item.

Parameter	Type/Description
hMenu	**HMENU** Identifies the menu to be changed.
nPosition	**WORD** Specifies the menu item to be changed. If *wFlags* is set to MF_BYPOSITION, *nPosition* specifies the position of the menu item; the first item in the menu is at position 0. If *wFlags* is set to MF_BYCOMMAND, then *nPosition* specifies the command ID of the menu item.
wFlags	**WORD** Specifies how the *nPosition* parameter is interpreted. It may be set to MF_BYCOMMAND (the default) or MF_BYPOSITION.
hBitmapUnchecked	**HBITMAP** Identifies the bitmap to be displayed when the menu item is not checked.

Parameter	Type/Description
hBitmapChecked	**HBITMAP** Identifies the bitmap to be displayed when the menu item is checked.

Return Value The return value specifies the outcome of the function. It is TRUE if the function is successful. Otherwise, it is FALSE.

Comments If either the *hBitmapUnchecked* or the *hBitmapChecked* parameters is NULL, then Windows displays nothing next to the menu item for the corresponding attribute. If both parameters are NULL, Windows uses the default checkmark when the item is checked and removes the checkmark when the item is unchecked.

When the menu is destroyed, these bitmaps are not destroyed; it is the responsibility of the application to destroy them.

The **GetMenuCheckMarkDimensions** function retrieves the dimensions of the default checkmark used for menu items. The application should use these values to determine the appropriate size for the bitmaps supplied with this function.

SetMessageQueue

Syntax BOOL SetMessageQueue(*cMsg*)

This function creates a new message queue. It is particularly useful in applications that require a queue that contains more than eight messages (the maximum size of the default queue). The *cMsg* parameter specifies the size of the new queue; the function must be called from an application's WinMain function before any windows are created and before any messages are sent. The **SetMessageQueue** function destroys the old queue, along with messages it might contain.

Parameter	Type/Description
cMsg	**int** Specifies the maximum number of messages that the new queue may contain.

S

Return Value The return value specifies whether a new message queue is created. It is nonzero if the function creates a new queue. Otherwise, it is zero.

Comments If the return value is zero, the application has no queue because the **SetMessageQueue** function deletes the original queue before attempting to create a new one. The application must continue calling **SetMessageQueue** with a smaller queue size until the function returns a nonzero value.

SetMetaFileBits

Syntax **HANDLE SetMetaFileBits**(*hMem*)

This function creates a memory metafile from the data in the global memory block specified by the *hMem* parameter.

Parameter	Type/Description
hMem	**HANDLE** Identifies the global memory block that contains the metafile data. It is assumed that the data were previously created by using the **GetMetaFileBits** function.

Return Value The return value identifies a memory metafile if the function is successful. Otherwise, the return value is NULL.

Comments After the **SetMetaFileBits** function returns, the metafile handle returned by the function should be used instead of the handle identified by the *hMem* parameter to refer to the metafile.

SetPaletteEntries 3.0

Syntax **WORD SetPaletteEntries**(*hPalette*, *wStartIndex*, *wNumEntries*, *lpPaletteEntries*)

This function sets RGB color values and flags in a range of entries in a logical palette.

Parameter	Type/Description
hPalette	**HPALETTE** Identifies the logical palette.
wStartIndex	**WORD** Specifies the first entry in the logical palette to be set.
wNumEntries	**WORD** Specifies the number of entries in the logical palette to be set.
lpPaletteEntries	**LPPALETTEENTRY** Points to the first member of an array of **PALETTEENTRY** data structures containing the RGB values and flags.

Return Value The return value is the number of entries set in the logical palette. It is zero if the function failed.

Comments If the logical palette is selected into a device context when the application calls **SetPalette-Entries**, the changes will not take effect until the application calls **RealizePalette**.

SetParent

Syntax **HWND SetParent**(*hWndChild*, *hWndNewParent*)

This function changes the parent window of a child window. If the window identified by the *hWndChild* parameter is visible, Windows performs the appropriate redrawing and re-painting.

Parameter	Type/Description
hWndChild	**HWND** Identifies the child window.
hWndNewParent	**HWND** Identifies the new parent window.

Return Value The return value identifies the previous parent window.

SetPixel

Syntax **DWORD SetPixel**(*hDC*, *X*, *Y*, *crColor*)

This function sets the pixel at the point specified by the *X* and *Y* parameters to the closest approximation of the color specified by the *crColor* parameter. The point must be in the clipping region. If the point is not in the clipping region, the function is ignored.

Parameter	Type/Description
hDC	**HDC** Identifies the device context.
X	**int** Specifies the logical *x*-coordinate of the point to be set.
Y	**int** Specifies the logical *y*-coordinate of the point to be set.
crColor	**COLORREF** Specifies the color used to paint the point.

Return Value The return value specifies an RGB color value for the color that the point is actually painted. This value can be different than that specified by the *crColor* parameter if an approximation of that color is used. If the function fails (if the point is outside the clipping region) the return value is –1.

Comments Not all devices support the **SetPixel** function. For more information, see the RC_BITBLT capability in the **GetDeviceCaps** function, earlier in this chapter.

SetPolyFillMode

Syntax **int SetPolyFillMode**(*hDC*, *nPolyFillMode*)

This function sets the polygon-filling mode for the GDI functions that use the polygon algorithm to compute interior points.

Parameter	Type/Description
hDC	**HDC** Identifies the device context.
nPolyFillMode	**int** Specifies the new filling mode. The *nPolyFillMode* parameter may be either of the following values:

Value	Meaning
ALTERNATE	Selects alternate mode.
WINDING	Selects winding number mode.

Return Value The return value specifies the previous filling mode. It is zero if there is an error.

Comments In general, the modes differ only in cases where a complex, overlapping polygon must be filled (for example, a five-sided polygon that forms a five-pointed star with a pentagon in the center). In such cases, ALTERNATE mode fills every other enclosed region within the polygon (that is, the points of the star), but WINDING mode fills all regions (that is, the points and the pentagon).

When the filling mode is ALTERNATE, GDI fills the area between odd-numbered and even-numbered polygon sides on each scan line. That is, GDI fills the area between the first and second side, between the third and fourth side, and so on.

To fill all regions, WINDING mode causes GDI to compute and draw a border that encloses the polygon but does not overlap. For example, in WINDING mode, the five-sided polygon that forms the star is drawn as a ten-sided polygon with no overlapping sides; the resulting star is filled.

SetProp

Syntax

BOOL **SetProp**(*hWnd*, *lpString*, *hData*)

This function adds a new entry or changes an existing entry in the property list of the specified window. The **SetProp** function adds a new entry to the list if the character string specified by the *lpString* parameter does not already exist in the list. The new entry contains the string and the handle. Otherwise, the function replaces the string's current handle with the one specified by the *hData* parameter.

The *hData* parameter can contain any 16-bit value useful to the application.

Parameter	Type/Description
hWnd	**HWND** Identifies the window whose property list is to receive the new entry.
lpString	**LPSTR** Points to a null-terminated character string or an atom that identifies a string. If an atom is given, it must have been previously created by using the **AddAtom** function. The atom, a 16-bit value, must be placed in the low-order word of *lpString*; the high-order word must be zero.
hData	**HANDLE** Identifies a data handle to be copied to the property list.

Return Value

The return value specifies the outcome of the function. It is nonzero if the data handle and string are added to the property list. Otherwise, it is zero.

Comments

The application is responsible for removing all entries it has added to the property list before destroying the window (that is, before the application processes the WM_DESTROY message). The **RemoveProp** function must be used to remove entries from a property list.

SetRect

Syntax

void **SetRect**(*lpRect*, *X1*, *Y1*, *X2*, *Y2*)

This function creates a new rectangle by filling the **RECT** data structure pointed to by the *lpRect* parameter with the coordinates given by the *X1*, *Y1*, *X2*, and *Y2* parameters.

Parameter	Type/Description
lpRect	**LPRECT** Points to the **RECT** data structure that is to receive the new rectangle coordinates.
X1	**int** Specifies the x-coordinate of the upper-left corner.
Y1	**int** Specifies the y-coordinate of the upper-left corner.
X2	**int** Specifies the x-coordinate of the lower-right corner.
Y2	**int** Specifies the y-coordinate of the lower-right corner.

Return Value None.

Comments The width of the rectangle, specified by the absolute value of $X2 - X1$, must not exceed 32,767 units. This limit applies to the height of the rectangle as well.

SetRectEmpty

Syntax **void SetRectEmpty(*lpRect*)**

This function creates an empty rectangle (all coordinates equal to zero).

Parameter	Type/Description
lpRect	**LPRECT** Points to the **RECT** data structure that is to receive the empty rectangle.

Return Value None.

SetRectRgn

Syntax **void SetRectRgn(*hRgn*, *X1*, *Y1*, *X2*, *Y2*)**

This function creates a rectangular region. Unlike **CreateRectRegion**, however, it does not call the local memory manager; instead, it uses the space allocated for the region associated with the *hRgn* parameter. The points given by the *X1*, *Y1*, *X2*, and *Y2* parameters specify the minimum size of the allocated space.

Parameter	Type/Description
hRgn	**HANDLE** Identifies the region.
X1	**int** Specifies the *x*-coordinate of the upper-left corner of the rectangular region.
Y1	**int** Specifies the *y*-coordinate of the upper-left corner of the rectangular region.
X2	**int** Specifies the *x*-coordinate of the lower-right corner of the rectangular region.
Y2	**int** Specifies the *y*-coordinate of the lower-right corner of the rectangular region.

Return Value None.

Comments Use this function instead of the **CreateRectRgn** function to avoid calls to the local memory manager.

SetResourceHandler

Syntax FARPROC **SetResourceHandler**(*hInstance*, *lpType*, *lpLoadFunc*)

This function sets up a function to load resources. It is used internally by Windows to implement calculated resources. Applications may find this function useful for handling their own resource types, but its use is not required. The *lpLoadFunc* parameter points to an application-supplied callback function. The function pointed to by the *lpLoadFunc* parameter receives information about the resource to be locked and can process that information as desired. After the function pointed to by *lpLoadFunc* returns, **LockResource** attempts to lock the resource once more.

Parameter	Type/Description
hInstance	**HANDLE** Identifies the instance of the module whose executable file contains the resource.
lpType	**LPSTR** Points to a short integer that specifies a resource type.
lpLoadFunc	**FARPROC** Is the procedure-instance address of the application-supplied callback function. See the following "Comments" section for details.

Return Value The return value points to the application-supplied function.

Comments The callback function must use the Pascal calling convention and must be declared **FAR**.

Callback Function **FARPROC FAR PASCAL** *LoadFunc*(*hMem*, *hInstance*, *hResInfo*)
HANDLE *hMem*;
HANDLE *hInstance*;
HANDLE *hResInfo*;

LoadFunc is a placeholder for the application-supplied function name. The actual name must be exported by including it in an **EXPORTS** statement in the application's module-definition file.

Parameter	Description
hMem	Identifies a stored resource.
hInstance	Identifies the instance of the module whose executable file contains the resource.
hResInfo	Identifies the resource. It is assumed that the resource was created previously by using the **FindResource** function.

Comments

The *hMem* parameter is NULL if the resource has not yet been loaded. If an attempt to lock a block specified by *hMem* fails, this means the resource has been discarded and must be reloaded.

The dialog-function address, passed as the *lpLoadFunc* parameter, must be created by using the **MakeProcInstance** function.

SetROP2

Syntax

int SetROP2(*hDC*, *nDrawMode*)

This function sets the current drawing mode. GDI uses the drawing mode to combine pens and interiors of filled objects with the colors already on the display surface. The mode specifies how the color of the pen or interior and the color already on the display surface yield a new color.

Parameter	Type/Description
hDC	**HDC** Identifies the device context.
nDrawMode	**int** Specifies the new drawing mode. It can be any one of the values given in Table 4.16, "Drawing Modes."

Return Value

The return value specifies the previous drawing mode. It can be any one of the values given in Chapter 11, "Binary and Ternary Raster-Operation Codes," in *Reference, Volume 2*.

Comments

Drawing modes define how GDI combines source and destination colors when drawing with the current pen. The drawing modes are actually binary raster-operation codes, representing all possible Boolean functions of two variables, using the binary operations AND, OR, and XOR (exclusive OR), and the unary operation NOT. The drawing mode is for raster devices only; it is not available on vector devices. For more information, see the RC_BITBLT capability in the **GetDeviceCaps** function, earlier in this chapter. Table 4.16 shows the value of various drawing modes for the *nDrawMode* parameter:

Table 4.16 Drawing Modes

Value	Meaning
R2_BLACK	Pixel is always black.
R2_WHITE	Pixel is always white.
R2_NOP	Pixel remains unchanged.
R2_NOT	Pixel is the inverse of the display color.
R2_COPYPEN	Pixel is the pen color.
R2_NOTCOPYPEN	Pixel is the inverse of the pen color.
R2_MERGEPENNOT	Pixel is a combination of the pen color and the inverse of the display color.
R2_MASKPENNOT	Pixel is a combination of the colors common to both the pen and the inverse of the display.
R2_MERGENOTPEN	Pixel is a combination of the display color and the inverse of the pen color.
R2_MASKNOTPEN	Pixel is a combination of the colors common to both the display and the inverse of the pen.
R2_MERGEPEN	Pixel is a combination of the pen color and the display color.
R2_NOTMERGEPEN	Pixel is the inverse of the R2_MERGEPEN color.
R2_MASKPEN	Pixel is a combination of the colors common to both the pen and the display.
R2_NOTMASKPEN	Pixel is the inverse of the R2_MASKPEN color.
R2_XORPEN	Pixel is a combination of the colors in the pen and in the display, but not in both.
R2_NOTXORPEN	Pixel is the inverse of the R2_XORPEN color.

For more information about the drawing modes, see Chapter 11, "Binary and Ternary Raster-Operation Codes," in *Reference, Volume 2.*

SetScrollPos

Syntax **int** **SetScrollPos**(*hWnd*, *nBar*, *nPos*, *bRedraw*)

This function sets the current position of a scroll-bar thumb to that specified by the *nPos* parameter and, if specified, redraws the scroll bar to reflect the new position.

Parameter	Type/Description
hWnd	**HWND** Identifies the window whose scroll bar is to be set.
nBar	**int** Specifies the scroll bar to be set. It can be one of the following values:

Value	Meaning
SB_CTL	Sets the position of a scroll-bar control. In this case, the *hWnd* parameter must be the handle of a scroll-bar control.
SB_HORZ	Sets a window's horizontal scroll-bar position.
SB_VERT	Sets a window's vertical scroll-bar position.

Parameter	Type/Description
nPos	**int** Specifies the new position. It must be within the scrolling range.
bRedraw	**BOOL** Specifies whether the scroll bar should be redrawn to reflect the new position. If the *bRedraw* parameter is nonzero, the scroll bar is redrawn. If it is zero, it is not redrawn.

Return Value The return value specifies the previous position of the scroll-bar thumb.

Comments Setting the *bRedraw* parameter to zero is useful whenever the scroll bar will be redrawn by a subsequent call to another function.

SetScrollRange

Syntax **void SetScrollRange**(*hWnd*, *nBar*, *nMinPos*, *nMaxPos*, *bRedraw*)

This function sets minimum and maximum position values for the given scroll bar. It can also be used to hide or show standard scroll bars by setting the *nMinPos* and *nMaxPos* parameters to zero.

Parameter	Type/Description
hWnd	**HWND** Identifies a window or a scroll-bar control, depending on the value of the *nBar* parameter.
nBar	**int** Specifies the scroll bar to be set. It can be one of the following values:

Value	Meaning
SB_CTL	Sets the range of a scroll-bar control. In this case, the *hWnd* parameter must be the handle of a scroll-bar control.
SB_HORZ	Sets a window's horizontal scroll-bar range.
SB_VERT	Sets a window's vertical scroll-bar range.

Parameter	Type/Description
nMinPos	**int** Specifies the minimum scrolling position.
nMaxPos	**int** Specifies the maximum scrolling position.
bRedraw	**BOOL** Specifies whether or not the scroll bar should be redrawn to reflect the change. If the *bRedraw* parameter is nonzero, the scroll bar is redrawn. If it is zero, it is not redrawn.

Return Value None.

Comments An application should not call this function to hide a scroll bar while processing a scroll-bar notification message.

If **SetScrollRange** immediately follows the **SetScrollPos** function, the *bRedraw* parameter in **SetScrollPos** should be set to zero to prevent the scroll bar from being drawn twice.

The difference between the values specified by the *nMinPos* and *nMaxPos* parameters must not be greater than 32,767.

SetSoundNoise

Syntax **int SetSoundNoise**(*nSource*, *nDuration*)

This function sets the source and duration of a noise in the noise hardware of the play device.

Parameter	Type/Description
nSource	**int** Specifies the noise source. It can be any one of the following values, where N is a value used to derive a target frequency:

Value	Meaning
S_PERIOD512	Source frequency is N/512 (high pitch); hiss is less coarse.
S_PERIOD1024	Source frequency is N/1024.
S_PERIOD2048	Source frequency is N/2048 (low pitch); hiss is coarser.
S_PERIODVOICE	Source frequency from voice channel 3.
S_WHITE512	Source frequency is N/512 (high pitch); hiss is less coarse.
S_WHITE1024	Source frequency is N/1024.
S_WHITE2048	Source frequency is N/2048 (low pitch); hiss is coarser.
S_WHITEVOICE	Source frequency from voice channel 3.

nDuration	**int** Specifies the duration of the noise (in clock ticks).

Return Value The return value specifies the result of the function. It is zero if the function is successful. If the source is invalid, the return value is S_SERDSR.

SetStretchBltMode

Syntax **int SetStretchBltMode**(*hDC*, *nStretchMode*)

This function sets the stretching mode for the **StretchBlt** function. The stretching mode defines which scan lines and/or columns **StretchBlt** eliminates when contracting a bitmap.

Parameter	Type/Description
hDC	**HDC** Identifies the device context.
nStretchMode	**int** Specifies the new stretching mode. It can be one of the following values:

Value	Meaning
BLACKONWHITE	AND in the *eliminated* lines. This mode preserves black pixels at the expense of white pixels by using the AND operator on the eliminated lines and those remaining.
COLORONCOLOR	Deletes the *eliminated* lines. This mode deletes all eliminated lines without trying to preserve their information.
WHITEONBLACK	OR in the *eliminated* lines. This mode preserves white pixels at the expense of black pixels by using the OR operator on the lines to be eliminated and the remaining lines.

The BLACKONWHITE and WHITEONBLACK modes are typically used to preserve foreground pixels in monochrome bitmaps. The COLORONCOLOR mode is typically used to preserve color in color bitmaps.

Return Value The return value specifies the previous stretching mode. It can be BLACKONWHITE, COLORONCOLOR, or WHITEONBLACK.

SetSwapAreaSize

Syntax **LONG SetSwapAreaSize**(*rsSize*)

This function increases the amount of memory that an application uses for its code segments. The maximum amount of memory available is one-half of the space remaining after Windows is loaded.

Parameter	Type/Description
rsSize	**WORD** Specifies the number of 16-byte paragraphs requested by the application for use as a code segment.

Return Value The low-order word of the return value specifies the number of paragraphs obtained for use as a code segment space (or the current size if *rsSize* is zero); the high-order word specifies the maximum size available.

Comments If *rsSize* specifies a size larger than is available, this function sets the size to the available amount.

Once memory has been dedicated for use as code segment space, an application cannot use it as a data segment by calling the **GlobalAlloc** function.

Calling this function improves an application's performance by helping prevent thrashing. However, it reduces the amount of memory available for data objects and can reduce the performance of other applications. Before calling **SetSwapAreaSize**, an application should call **GetNumTasks** to determine how many other tasks are running.

SetSysColors

Syntax **void SetSysColors**(*nChanges*, *lpSysColor*, *lpColorValues*)

This function sets the system colors for one or more display elements. Display elements are the various parts of a window and the Windows display that appear on the system display screen. The **SetSysColors** function changes the number of elements specified by the *nChanges* parameter, using the color and system-color index contained in the arrays pointed to by the *lpSysColor* and *lpColorValues* parameters.

SetSysColors sends a WM_SYSCOLORCHANGE message to all windows to inform them of the change in color. It also directs Windows to repaint the affected portions of all currently visible windows.

Parameter	Type/Description
nChanges	**int** Specifies the number of system colors to be changed.
lpSysColor	**LPINT** Points to an array of integer indexes that specify the elements to be changed. The index values that can be used are listed in Table 4.17, "System Color Indexes."
lpColorValues	**DWORD FAR *** Points to an array of unsigned long integers that contains the new RGB color values for each element.

Return Value None.

Comments **SetSysColors** changes the internal system list only. It does not change the [COLORS] section of the Windows initialization file, WIN.INI. Changes apply to the current Windows session only. System colors are a shared resource. An application should not change a color if it does not wish to change colors for all windows in all currently running applications. System colors for monochrome displays are usually interpreted as various shades of gray. Table 4.17 lists the values for the *lpSysColor* parameter:

Table 4.17 System Color Indexes

Value	Meaning
COLOR_ACTIVEBORDER	Active window border.
COLOR_ACTIVECAPTION	Active window caption.
COLOR_APPWORKSPACE	Background color of multiple document interface (MDI) applications.
COLOR_BACKGROUND	Desktop.
COLOR_BTNFACE	Face shading on push buttons.
COLOR_BTNSHADOW	Edge shading on push buttons.
COLOR_BTNTEXT	Text on push buttons.
COLOR_CAPTIONTEXT	Text in caption, size box, scroll-bar arrow box.
COLOR_GRAYTEXT	Grayed (disabled) text. This color is set to 0 if the current display driver does not support a solid gray color.
COLOR_HIGHLIGHT	Items selected item in a control.
COLOR_HIGHLIGHTTEXT	Text of item selected in a control.
COLOR_INACTIVEBORDER	Inactive window border.
COLOR_INACTIVECAPTION	Inactive window caption.
COLOR_MENU	Menu background.
COLOR_MENUTEXT	Text in menus.
COLOR_SCROLLBAR	Scroll-bar gray area.
COLOR_WINDOW	Window background.
COLOR_WINDOWFRAME	Window frame.
COLOR_WINDOWTEXT	Text in windows.

SetSysModalWindow

Syntax **HWND SetSysModalWindow**(*hWnd*)

This function makes the specified window a system-modal window.

Parameter	Type/Description
hWnd	**HWND** Identifies the window to be made system modal.

Return Value The return value identifies the window that was previously the system-modal window.

Comments If another window is made the active window (for example, the system-modal window creates a dialog box that becomes the active window), the active window becomes the system-modal window. When the original window becomes active again, it is system modal. To end the system-modal state, destroy the system-modal window.

SetSystemPaletteUse 3.0

Syntax **WORD SetSystemPaletteUse**(*hDC*, *wUsage*)

This function allows an application to use the full system palette. By default, the system palette contains 20 static colors which are not changed when an application realizes its logical palette.

The device context identified by the *hDC* parameter must refer to a device that supports color palettes.

Parameter	Type/Description
hDC	**HDC** Identifies the device context.
wUsage	**WORD** Specifies the new use of the system palette. It can be either of these values:

Value	Meaning
SYSPAL_NOSTATIC	System palette contains no static colors except black and white.
SYSPAL_STATIC	System palette contains static colors which will not change when an application realizes its logical palette.

Return Value The return value specifies the previous usage of the system palette. It is either SYSPAL_NOSTATIC or SYSPAL_STATIC.

Comments An application must call this function only when its window has input focus.

If an application calls **SetSystemPaletteUse** with *wUsage* set to SYSPAL_NOSTATIC, Windows continues to set aside two entries in the system palette for pure white and pure black, respectively.

After calling this function with *wUsage* set to SYSPAL_NOSTATIC, an application must follow these steps:

1. Call **UnrealizeObject** to force GDI to remap the logical palette completely when it is realized.
2. Realize the logical palette.
3. Call **GetSysColors** to save the current system-color settings.
4. Call **SetSysColors** to set the system colors to reasonable values using black and white. For example, adjacent or overlapping items (such as window frames and borders) should be set to black and white, respectively.
5. Broadcast the WM_SYSCOLORCHANGE message to allow other windows to be redrawn with the new system colors.

When the application's window loses focus or closes, the application must perform the following steps:

1. Call **SetSystemPaletteUse** with the *wUsage* parameter set to SYSPAL_STATIC.
2. Call **UnrealizeObject** to force GDI to remap the logical palette completely when it is realized.
3. Realize the logical palette.
4. Restore the system colors to their previous values.
5. Broadcast the WM_SYSCOLORCHANGE message.

SetTextAlign

Syntax **WORD** **SetTextAlign**(*hDC*, *wFlags*)

This function sets the text-alignment flags for the given device context. The **TextOut** and **ExtTextOut** functions use these flags when positioning a string of text on a display or device. The flags specify the relationship between a specific point and a rectangle that bounds the text. The coordinates of this point are passed as parameters to the **TextOut** function. The rectangle that bounds the text is formed by the adjacent character cells in the text string.

Parameter	Type/Description
hDC	**HDC** Identifies the device or display selected for text output.
wFlags	**WORD** Specifies a mask of the values in the following list. Only one flag may be chosen from those that affect horizontal and vertical alignment. In addition, only one of the two flags that alter the current position can be chosen:

Value	Meaning
TA_BASELINE	Specifies alignment of the point and the baseline of the chosen font.
TA_BOTTOM	Specifies alignment of the point and the bottom of the bounding rectangle.
TA_CENTER	Specifies alignment of the point and the horizontal center of the bounding rectangle.
TA_LEFT	Specifies alignment of the point and the left side of the bounding rectangle.
TA_NOUPDATECP	Specifies that the current position is not updated after each **TextOut** or **ExtTextOut** function call.
TA_RIGHT	Specifies alignment of the point and the right side of the bounding rectangle.
TA_TOP	Specifies alignment of the point and the top of the bounding rectangle.
TA_UPDATECP	Specifies that the current position is updated after each **TextOut** or **ExtTextOut** function call.

The defaults are TA_LEFT, TA_TOP, and TA_NOUPDATECP.

Return Value The return value specifies the previous text alignment setting; the low-order word contains the horizontal alignment, and the high-order word contains the vertical alignment.

SetTextCharacterExtra

Syntax **int SetTextCharacterExtra**(*hDC*, *nCharExtra*)

This function sets the amount of intercharacter spacing. GDI adds this spacing to each character, including break characters, when it writes a line of text to the device context.

Parameter	Type/Description
hDC	**HDC** Identifies the device context.
nCharExtra	**int** Specifies the amount of extra space (in logical units) to be added to each character. If the current mapping mode is not MM_TEXT, the *nCharExtra* parameter is transformed and rounded to the nearest pixel.

Return Value The return value specifies the amount of the previous intercharacter spacing.

SetTextColor

Syntax **DWORD SetTextColor**(*hDC*, *crColor*)

This function sets the text color to the color specified by the *crColor* parameter, or to the nearest physical color if the device cannot represent the color specified by *crColor*. GDI uses the text color to draw the face of each character written by the **TextOut** and **ExtTextOut** functions. GDI also uses the text color when converting bitmaps from color to monochrome and vice versa.

The background color for a character is specified by the **SetBkColor** and **SetBkMode** functions. For color-bitmap conversions, see the **BitBlt** and **StretchBlt** functions, earlier in this chapter.

Parameter	Type/Description
hDC	**HDC** Identifies the device context.
crColor	**COLORREF** Specifies the color of the text.

Return Value The return value specifies an RGB color value for the previous text color.

SetTextJustification

Syntax **int SetTextJustification**(*hDC*, *nBreakExtra*, *nBreakCount*)

This function prepares GDI to justify a line of text using the justification parameters specified by the *nBreakExtra* and *nBreakCount* parameters. To justify text, GDI distributes extra pixels among break characters in a text line written by the **TextOut** function. The break character, used to delimit words, is usually the space character (ASCII 32), but may be defined by a font as some other character. The **GetTextMetrics** function can be used to retrieve a font's break character.

The **SetTextJustification** function prepares the justification by defining the amount of space to be added. The *nBreakExtra* parameter specifies the total amount of space (in logical units) to be added to the line. The *nBreakCount* parameter specifies how many break characters are in the line. The subsequent **TextOut** function distributes the extra space evenly between each break character in the line.

GetTextExtent is always used with the **SetTextJustification** function. The **GetTextExtent** function computes the width of a given line before justification. This width must be known before an appropriate *nBreakExtra* value can be computed.

SetTextJustification can be used to justify a line that contains multiple runs in different fonts. In this case, the line must be created piecemeal by justifying and writing each run separately.

Because rounding errors can occur during justification, GDI keeps a running error term that defines the current error. When justifying a line that contains multiple runs, **GetTextExtent** automatically uses this error term when it computes the extent of the next run, allowing **TextOut** to blend the error into the new run. After each line has been justified, this error term must be cleared to prevent it from being incorporated into the next line. The term can be cleared by calling **SetTextJustification** with *nBreakExtra* set to zero.

Parameter	Type/Description
hDC	**HDC** Identifies the device context.
nBreakExtra	**int** Specifies the total extra space (in logical units) to be added to the line of text. If the current mapping mode is not MM_TEXT, the value identified by the *nBreakExtra* parameter is transformed and rounded to the nearest pixel.
nBreakCount	**int** Specifies the number of break characters in the line.

Return Value The return value specifies the outcome of the function. It is 1 if the function is successful. Otherwise, it is zero.

SetTimer

Syntax

WORD **SetTimer**(*hWnd*, *nIDEvent*, *wElapse*, *lpTimerFunc*)

This function creates a system timer event. When a timer event occurs, Windows passes a WM_TIMER message to the application-supplied function specified by the *lpTimerFunc* parameter. The function can then process the event. A NULL value for *lpTimerFunc* causes WM_TIMER messages to be placed in the application queue.

Parameter	Type/Description
hWnd	**HWND** Identifies the window to be associated with the timer. If *hWnd* is NULL, no window is associated with the timer.
nIDEvent	**int** Specifies a nonzero timer-event identifier if the *hWnd* parameter is not NULL.
wElapse	**WORD** Specifies the elapsed time (in milliseconds) between timer events.
lpTimerFunc	**FARPROC** Is the procedure-instance address of the function to be notified when the timer event takes place. If *lpTimerFunc* is NULL, the WM_TIMER message is placed in the application queue, and the **hwnd** member of the **MSG** structure contains the *hWnd* parameter given in the **SetTimer** function call. See the following "Comments" section for details.

Return Value

The return value specifies the integer identifier for the new timer event. If the *hWnd* parameter is NULL, an application passes this value to the **KillTimer** function to kill the timer event. The return value is zero if the timer was not created.

Comments

Timers are a limited global resource; therefore, it is important that an application check the value returned by the **SetTimer** function to verify that a timer is actually available.

To install a timer function, **SetTimer** must receive a procedure-instance address of the function, and the function must be exported in the application's module-definition file. A procedure-instance address can be created using the **MakeProcInstance** function.

The callback function must use the Pascal calling convention and must be declared **FAR**.

Callback Function

WORD FAR PASCAL *TimerFunc*(*hWnd*, *wMsg*, *nIDEvent*, *dwTime*)
HWND *hWnd*;
WORD *wMsg*;
int *nIDEvent*;
DWORD *dwTime*;

TimerFunc is a placeholder for the application-supplied function name. The actual name must be exported by including it in an **EXPORTS** statement in the application's module-definition file.

Parameter	Description
hWnd	Identifies the window associated with the timer event.
wMsg	Specifies the WM_TIMER message.
nIDEvent	Specifies the timer's ID.
dwTime	Specifies the current system time.

SetViewportExt

Syntax **DWORD SetViewportExt**(*hDC*, *X*, *Y*)

This function sets the *x*- and *y*-extents of the viewport of the specified device context. The viewport, along with the device-context window, defines how GDI maps points in the logical coordinate system to points in the coordinate system of the actual device. In other words, they define how GDI converts logical coordinates into device coordinates.

The *x*- and *y*-extents of the viewport define how much GDI must stretch or compress units in the logical coordinate system to fit units in the device coordinate system. For example, if the *x*-extent of the window is 2 and the *x*-extent of the viewport is 4, GDI maps two logical units (measured from the *x*-axis) into four device units. Similarly, if the *y*-extent of the window is 2 and the *y*-extent of the viewport is –1, GDI maps two logical units (measured from the *y*-axis) into one device unit.

The extents also define the relative orientation of the *x*- and *y*-axes in both coordinate systems. If the signs of matching window and viewport extents are the same, the axes have the same orientation. If the signs are different, the orientation is reversed. For example, if the *y*-extent of the window is 2 and the *y*-extent of the viewport is –1, GDI maps the positive *y*-axis in the logical coordinate system to the negative *y*-axis in the device coordinate system. If the *x*-extents are 2 and 4, GDI maps the positive *x*-axis in the logical coordinate system to the positive *x*-axis in the device-coordinate system.

Parameter	Type/Description
hDC	**HDC** Identifies the device context.
X	**int** Specifies the *x*-extent of the viewport (in device units).
Y	**int** Specifies the *y*-extent of the viewport (in device units).

Return Value The return value specifies the previous extents of the viewport. The previous *y*-extent is in the high-order word; the previous *x*-extent is in the low-order word. When an error occurs, the return value is zero.

Comments When the following mapping modes are set, calls to the **SetWindowExt** and **SetViewportExt** functions are ignored:

- MM_HIENGLISH
- MM_HIMETRIC
- MM_LOENGLISH
- MM_LOMETRIC
- MM_TEXT
- MM_TWIPS

When MM_ISOTROPIC mode is set, an application must call the **SetWindowExt** function before it calls **SetViewportExt**.

SetViewportOrg

Syntax **DWORD SetViewportOrg**(*hDC*, *X*, *Y*)

This function sets the viewport origin of the specified device context. The viewport, along with the device-context window, defines how GDI maps points in the logical coordinate system to points in the coordinate system of the actual device. In other words, they define how GDI converts logical coordinates into device coordinates.

The viewport origin marks the point in the device coordinate system to which GDI maps the window origin, a point in the logical coordinate system specified by the **SetWindowOrg** function. GDI maps all other points by following the same process required to map the window origin to the viewport origin. For example, all points in a circle around the point at the window origin will be in a circle around the point at the viewport origin. Similarly, all points in a line that passes through the window origin will be in a line that passes through the viewport origin.

Parameter	Type/Description
hDC	**HDC** Identifies the device context.
X	**int** Specifies the *x*-coordinate (in device units) of the origin of the viewport. The value must be within the range of the device coordinate system.

Parameter	Type/Description
Y	**int** Specifies the *y*-coordinate (in device units) of the origin of the viewport. The value must be within the range of the device coordinate system.

Return Value The return value specifies the previous origin of the viewport (in device coordinates). The *y*-coordinate is in the high-order word; the *x*-coordinate is in the low-order word.

SetVoiceAccent

Syntax **int SetVoiceAccent**(*nVoice*, *nTempo*, *nVolume*, *nMode*, *nPitch*)

This function places an accent (tempo, volume, mode, and pitch) in the voice queue specified by the *nVoice* parameter. The new accent replaces the previous accent and remains in effect until another accent is queued. An accent is not counted as a note.

An error occurs if there is insufficient room in the queue; the **SetVoiceAccent** function always leaves space for a single sync mark in the queue. If *nVoice* is out of range, the **SetVoiceAccent** function is ignored.

Parameter	Type/Description
nVoice	**int** Specifies a voice queue. The first voice queue is numbered 1.
nTempo	**int** Specifies the number of quarter notes played per minute. It can be any value from 32 to 255. The default is 120.
nVolume	**int** Specifies the volume level. It can be any value from 0 (lowest volume) to 255 (highest).
nMode	**int** Specifies how the notes are to be played. It can be any one of the following values:

Value	Meaning
S_LEGATO	Note is held for the full duration and blended with the beginning of the next note.
S_NORMAL	Note is held for the full duration, coming to a full stop before the next note starts.
S_STACCATO	Note is held for only part of the duration, creating a pronounced stop between it and the next note.

Parameter	Type/Description
nPitch	**int** Specifies the pitch of the notes to be played. It can be any value from 0 to 83. The pitch value is added to the note value, using modulo 84 arithmetic.

Return Value The return value specifies the result of the function. It is zero if the function is successful. If an error occurs, the return value is one of the following values:

Value	Meaning
S_SERDMD	Invalid mode
S_SERDTP	Invalid tempo
S_SERDVL	Invalid volume
S_SERQFUL	Queue full

SetVoiceEnvelope

Syntax **int SetVoiceEnvelope**(*nVoice*, *nShape*, *nRepeat*)

This function queues the envelope (wave shape and repeat count) in the voice queue specified by the *nVoice* parameter. The new envelope replaces the previous one and remains in effect until the next **SetVoiceEnvelope** function call. An envelope is not counted as a note.

An error occurs if there is insufficient room in the queue; the **SetVoiceEnvelope** function always leaves space for a single sync mark in the queue. If *nVoice* is out of range, **SetVoiceEnvelope** is ignored.

Parameter	Type/Description
nVoice	**int** Specifies the voice queue to receive the envelope.
nShape	**int** Specifies an index to an OEM wave-shape table.
nRepeat	**int** Specifies the number of repetitions of the wave shape during the duration of one note.

Return Value The return value specifies the result of the function. It is zero if the function is successful. If an error occurs, the return value is one of the following values:

Value	Meaning
S_SERDRC	Invalid repeat count
S_SERDSH	Invalid shape
S_SERQFUL	Queue full

SetVoiceNote

Syntax **int** **SetVoiceNote**(*nVoice*, *nValue*, *nLength*, *nCdots*)

This function queues a note that has the qualities given by the *nValue*, *nLength*, and *nCdots* parameters in the voice queue specified by the *nVoice* parameter. An error occurs if there is insufficient room in the queue. The function always leaves space in the queue for a single sync mark.

Parameter	Type/Description
nVoice	**int** Specifies the voice queue to receive the note. If *nVoice* is out of range, the **SetVoiceNote** function is ignored.
nValue	**int** Specifies 1 of 84 possible notes (seven octaves). If *nValue* is zero, a rest is assumed.
nLength	**int** Specifies the reciprocal of the duration of the note. For example, 1 specifies a whole note, 2 a half note, 4 a quarter note, and so on.
nCdots	**int** Specifies the duration of the note in dots. The duration is equal to *nLength* × (*nCdots* × 3/2).

Return Value The return value specifies the result of the function. It is zero if the function is successful. If an error occurs, the return value is one of the following values:

Value	Meaning
S_SERDCC	Invalid dot count
S_SERDLN	Invalid note length
S_SERDNT	Invalid note
S_SERQFUL	Queue full

SetVoiceQueueSize

Syntax **int** **SetVoiceQueueSize**(*nVoice*, *nBytes*)

This function allocates the number of bytes specified by the *nBytes* parameter for the voice queue specified by the *nVoice* parameter. If the queue size is not set, the default is 192 bytes, which is room for about 32 notes. All voice queues are locked in memory. The queues cannot be set while music is playing.

Parameter	Type/Description
nVoice	**int** Specifies a voice queue.
nBytes	**int** Specifies the number of bytes in the voice queue.

Return Value The return value specifies the result of the function. It is zero if the function is successful. If an error occurs, the return value is one of the following values:

Value	Meaning
S_SERMACT	Music active
S_SEROFM	Out of memory

SetVoiceSound

Syntax **int** **SetVoiceSound**(*nVoice*, *lFrequency*, *nDuration*)

This function queues the sound frequency and duration in the voice queue specified by the *nVoice* parameter.

Parameter	Type/Description
nVoice	**int** Specifies a voice queue. The first voice queue is numbered 1.
lFrequency	**long** Specifies the frequency. The high-order word contains the frequency in hertz, and the low-order word contains the fractional frequency.
nDuration	**int** Specifies the duration of the sound (in clock ticks).

Return Value The return value specifies the result of the function. It is zero if the function is successful. If an error occurs, the return value is one of the following values:

Value	Meaning
S_SERDDR	Invalid duration
S_SERDFQ	Invalid frequency
S_SERDVL	Invalid volume
S_SERQFUL	Queue full

SetVoiceThreshold

Syntax **int SetVoiceThreshold**(*nVoice*, *nNotes*)

This function sets the threshold level for the given voice. When the number of notes remaining in the voice queue goes below that specified in the *nNotes* parameter, the threshold flag is set. If the queue level is below that specified in *nNotes* when the **SetVoiceThreshold** function is called, the flag is not set. The **GetThresholdStatus** function should be called to verify the current threshold status.

Parameter	Type/Description
nVoice	**int** Specifies the voice queue to be set.
nNotes	**int** Specifies the number of notes in the threshold level.

Return Value The return value specifies the result of the function. It is zero if the function is successful. It is 1 if the number of notes specified in *nNotes* is out of range.

SetWindowExt

Syntax **DWORD SetWindowExt**(*hDC*, *X*, *Y*)

This function sets the *x*- and *y*-extents of the window associated with the specified device context. The window, along with the device-context viewport, defines how GDI maps points in the logical coordinate system to points in the device coordinate system.

The *x*- and *y*-extents of the window define how much GDI must stretch or compress units in the logical coordinate system to fit units in the device coordinate system. For example, if the *x*-extent of the window is 2 and the *x*-extent of the viewport is 4, GDI maps two logical units (measured from the *x*-axis) into four device units. Similarly, if the *y*-extent of the window is 2 and the *y*-extent of the viewport is –1, GDI maps two logical units (measured from the *y*-axis) into one device unit.

The extents also define the relative orientation of the *x*- and *y*-axes in both coordinate systems. If the signs of matching window and viewport extents are the same, the axes have the same orientation. If the signs are different, the orientation is reversed. For example, if the *y*-extent of the window is 2 and the *y*-extent of the viewport is –1, GDI maps the positive *y*-axis in the logical coordinate system to the negative *y*-axis in the device coordinate system. If the *x*-extents are 2 and 4, GDI maps the positive *x*-axis in the logical coordinate system to the positive *x*-axis in the device coordinate system.

Parameter	Type/Description
hDC	**HDC** Identifies the device context.
X	**int** Specifies the *x*-extent (in logical units) of the window.
Y	**int** Specifies the *y*-extent (in logical units) of the window.

Return Value The return value specifies the previous extents of the window (in logical units). The *y*-extent is in the high-order word; the *x*-extent is in the low-order word. If an error occurs, the return value is zero.

Comments When the following mapping modes are set, calls to the **SetWindowExt** and **SetViewportExt** functions are ignored:

- MM_HIENGLISH
- MM_HIMETRIC
- MM_LOENGLISH
- MM_LOMETRIC
- MM_TEXT
- MM_TWIPS

When MM_ISOTROPIC mode is set, an application must call the **SetWindowExt** function before calling **SetViewportExt**.

SetWindowLong

Syntax LONG **SetWindowLong**(*hWnd*, *nIndex*, *dwNewLong*)

This function changes an attribute of the window specified by the *hWnd* parameter.

Parameter	Type/Description
hWnd	**HWND** Identifies the window.
nIndex	**int** Specifies the byte offset of the attribute to be changed. It may also be one of the following values:

Value	Meaning
GWL_EXSTYLE	Sets a new extended window style.
GWL_STYLE	Sets a new window style.
GWL_WNDPROC	Sets a new long pointer to the window procedure.

dwNewLong	**DWORD** Specifies the replacement value.

Return Value The return value specifies the previous value of the specified long integer.

Comments If the **SetWindowLong** function and the GWL_WNDPROC index are used to set a new window function, that function must have the window-function form and be exported in the module-definition file of the application. For more information, see the **RegisterClass** function, earlier in this chapter.

Calling **SetWindowLong** with the GCL_WNDPROC index creates a subclass of the window class used to create the window. See Chapter 1, "Window Manager Interface Functions," for more information on window subclassing. An application should not attempt to create a window subclass for standard Windows controls such as combo boxes and buttons.

To access any extra four-byte values allocated when the window-class structure was created, use a positive byte offset as the index specified by the *nIndex* parameter, starting at zero for the first four-byte value in the extra space, 4 for the next four-byte value and so on.

SetWindowOrg

Syntax **DWORD SetWindowOrg**(*hDC*, *X*, *Y*)

This function sets the window origin of the specified device context. The window, along with the device-context viewport, defines how GDI maps points in the logical coordinate system to points in the device coordinate system.

The window origin marks the point in the logical coordinate system from which GDI maps the viewport origin, a point in the device coordinate system specified by the **SetWindow-**

Org function. GDI maps all other points by following the same process required to map the window origin to the viewport origin. For example, all points in a circle around the point at the window origin will be in a circle around the point at the viewport origin. Similarly, all points in a line that passes through the window origin will be in a line that passes through the viewport origin.

Parameter	Type/Description
hDC	**HDC** Identifies the device context.
X	**int** Specifies the logical *x*-coordinate of the new origin of the window.
Y	**int** Specifies the logical *y*-coordinate of the new origin of the window.

Return Value The return value specifies the previous origin of the window. The previous *y*-coordinate is in the high-order word; the previous *x*-coordinate is in the low-order word.

SetWindowPos

Syntax **void SetWindowPos**(*hWnd*, *hWndInsertAfter*, *X*, *Y*, *cx*, *cy*, *wFlags*)

This function changes the size, position, and ordering of child, pop-up, and top-level windows. Child, pop-up, and top-level windows are ordered according to their appearance on the screen; the topmost window receives the highest rank, and it is the first window in the list. This ordering is recorded in a window list.

Parameter	Type/Description
hWnd	**HWND** Identifies the window that will be positioned.
hWndInsertAfter	**HWND** Identifies a window in the window-manager's list that will precede the positioned window.
X	**int** Specifies the *x*-coordinate of the window's upper-left corner.
Y	**int** Specifies the *y*-coordinate of the window's upper-left corner.
cx	**int** Specifies the new window's width.
cy	**int** Specifies the new window's height.

Parameter	Type/Description
wFlags	**WORD** Specifies one of eight possible 16-bit values that affect the sizing and positioning of the window. It must be one of the following values:

Value	Meaning
SWP_DRAWFRAME	Draws a frame (defined in the window's class description) around the window.
SWP_HIDEWINDOW	Hides the window.
SWP_NOACTIVATE	Does not activate the window.
SWP_NOMOVE	Retains current position (ignores the *x* and *y* parameters).
SWP_NOSIZE	Retains current size (ignores the *cx* and *cy* parameters).
SWP_NOREDRAW	Does not redraw changes.
SWP_NOZORDER	Retains current ordering (ignores the *hWndInsertAfter* parameter).
SWP_SHOWWINDOW	Displays the window.

Return Value

None.

Comments

If the SWP_NOZORDER flag is not specified, Windows places the window identified by the *hWnd* parameter in the position following the window identified by the *hWndInsertAfter* parameter. If *hWndInsertAfter* is NULL, Windows places the window identified by *hWnd* at the top of the list. If *hWndInsertAfter* is set to 1, Windows places the window identified by *hWnd* at the bottom of the list.

If the SWP_SHOWWINDOW or the SWP_HIDEWINDOW flags are set, scrolling and moving cannot be done simultaneously.

All coordinates for child windows are relative to the upper-left corner of the parent window's client area.

SetWindowsHook

Syntax **FARPROC SetWindowsHook**(*nFilterType, lpFilterFunc*)

This function installs a filter function in a chain. A filter function processes events before they are sent to an application's message loop in the WinMain function. A chain is a linked list of filter functions of the same type.

Parameter	Type/Description
nFilterType	**int** Specifies the system hook to be installed. It can be any one of the following values:

Value	Meaning
WH_CALLWNDPROC	Installs a window-function filter.
WH_GETMESSAGE	Installs a message filter.
WH_JOURNALPLAYBACK	Installs a journaling playback filter.
WH_JOURNALRECORD	Installs a journaling record filter.
WH_KEYBOARD	Installs a keyboard filter.
WH_MSGFILTER	Installs a message filter.
WH_SYSMSGFILTER	Installs a system-wide message filter.

Parameter	Type/Description
lpFilterFunc	**FARPROC** Is the procedure-instance address of the filter function to be installed. See the following "Comments" section for details.

Return Value The return value points to the procedure-instance address of the previously installed filter (if any). It is NULL if there is no previous filter. The application or library that calls the **SetWindowsHook** function should save this return value in the library's data segment. The fourth argument of the **DefHookProc** function points to the location in memory where the library saves this return value.

Comments The WH_CALLWNDPROC hook will affect system performance. It is supplied for debugging purposes only.

The system hooks are a shared resource. Installing a hook affects all applications. Most hook functions must be in libraries. The only exception is WH_MSGFILTER, which is task-specific. System hooks should be restricted to special-purpose applications or as a development aid during debugging of an application. Libraries that no longer need the hook should remove the filter function.

To install a filter function, the **SetWindowsHook** function must receive a procedure-instance address of the function, and the function must be exported in the library's module-definition file. Libraries can pass the procedure address directly. Tasks must use **MakeProcInstance** to get a procedure-instance address. Dynamic-link libraries must use **GetProcAddress** to get a procedure-instance address.

The following section describes how to support the individual hook functions.

WH_CALLWNDPROC

Windows calls the WH_CALLWNDPROC filter function whenever the **SendMessage** function is called. Windows does not call the filter function when the **PostMessage** function is called.

The filter function must use the Pascal calling convention and must be declared **FAR**. The filter function must have the following form:

Filter Function

void FAR PASCAL *FilterFunc*(*nCode*, *wParam*, *lParam*)
int *nCode*;
WORD *wParam*;
DWORD *lParam*;

FilterFunc is a placeholder for the application- or library-supplied function name. The actual name must be exported by including it in an **EXPORTS** statement in the library's module-definition file.

Parameter	Description
nCode	Specifies whether the filter function should process the message or call the **DefHookProc** function. If the *nCode* parameter is less than zero, the filter function should pass the message to **DefHookProc** without further processing.
wParam	Specifies whether the message is sent by the current task. It is nonzero if the message is sent; otherwise, it is NULL.
lParam	Points to a data structure that contains details about the message intercepted by the filter. The following shows the order, type, and description of each field of the data structure:

Parameter	Description	
	Field	**Type/Description**
	hlParam	**WORD** Contains the high-order word of the *lParam* parameter of the message received by the filter.
	llParam	**WORD** Contains the low-order word of the *lParam* parameter of the message received by the filter.
	wParam	**WORD** Contains the *wParam* parameter of the message received by the filter.
	wMsg	**WORD** Contains the message received by the filter.
	hWnd	**WORD** Contains the window handle of the window that is to receive the message.

Comments

The WH_CALLWNDPROC filter function can examine or modify the message as desired. Once it returns control to Windows, the message, with any modifications, is passed on to the window function. The filter function does not require a return value.

WH_GETMESSAGE

Windows calls the WH_GETMESSAGE filter function whenever the **GetMessage** function is called. Windows calls the filter function immediately after **GetMessage** has retrieved a message from an application queue. The filter function must use the Pascal calling convention and must be declared **FAR**. The filter function must have the following form:

Filter Function

void FAR PASCAL *FilterFunc*(*nCode*, *wParam*, *lParam*)
int *nCode*;
WORD *wParam*;
DWORD *lParam*;

FilterFunc is a placeholder for the library-supplied function name. The actual name must be exported by including it in an **EXPORTS** statement in the library's module-definition file.

Parameter	Description
nCode	Specifies whether the filter function should process the message or call the **DefHookProc** function. If the *nCode* parameter is less than zero, the filter function should pass the message to **DefHookProc** without further processing.
wParam	Specifies a NULL value.
lParam	Points to a message structure.

Comments

The WH_GETMESSAGE filter function can examine or modify the message as desired. Once it returns control to Windows, the **GetMessage** function returns the message, with any modifications, to the application that originally called it. The filter function does not require a return value.

WH_JOURNALPLAYBACK

Windows calls the WH_JOURNALPLAYBACK filter function whenever a request for an event message is made. The function is intended to be used to supply a previously recorded event message.

The filter function must use the Pascal calling convention and must be declared **FAR**. The filter function must have the following form:

Filter Function

DWORD FAR PASCAL *FilterFunc*(*nCode*, *wParam*, *lParam*);
int *nCode*;
WORD *wParam*;
DWORD *lParam*;

FilterFunc is a placeholder for the library-supplied function name. The actual name must be exported by including it in an **EXPORTS** statement in the library's module-definition file.

S

Parameter	Description
nCode	Specifies whether the filter function should process the message or call the **DefHookProc** function. If the *nCode* parameter is less then zero, the filter function should pass the message to **DefHookProc** without further processing.

Parameter	Description
wParam	Specifies a NULL value.
lParam	Points to the message being processed by the filter function.

Comments

The WH_JOURNALPLAYBACK function should copy an event message to the *lParam* parameter. The message must have been previously recorded by using the WH_JOURNALRECORD filter. It should not modify the message. The return value should be the amount of time (in clock ticks) Windows should wait before processing the message. This time can be computed by calculating the difference between the **time** fields in the current and previous event messages. If the function returns zero, the message is processed immediately. Once it returns control to Windows, the message continues to be processed. If the *nCode* parameter is HC_SKIP, the filter function should prepare to return the next recorded event message on its next call.

While the WH_JOURNALPLAYBACK function is in effect, Windows ignores all mouse and keyboard input.

WH_JOURNALRECORD

Windows calls the WH_JOURNALRECORD filter function whenever it processes a message from the event queue. The filter can be used to record the event for later playback.

The filter function must use the Pascal calling convention and must be declared **FAR**. The filter function must have the following form:

Filter Function

void FAR PASCAL *FilterFunc*(*nCode*, *wParam*, *lParam*);
int *nCode*;
WORD *wParam*;
DWORD *lParam*;

FilterFunc is a placeholder for the library-supplied function name. The actual name must be exported by including it in an **EXPORTS** statement in the library's module-definition file.

Parameter	Description
nCode	Specifies whether the filter function should process the message or call the **DefHookProc** function. If the *nCode* parameter is less than zero, the filter function should pass the message to **DefHookProc** without further processing.

Parameter	Description
wParam	Specifies a NULL value.
lParam	Points to a message structure.

Comments

The WH_JOURNALRECORD function should save a copy of the event message for later playback. It should not modify the message. Once it returns control to Windows, the message continues to be processed. The filter function does not require a return value.

WH_KEYBOARD

Windows calls the WH_KEYBOARD filter function whenever the application calls the **GetMessage** or **PeekMessage** function and there is a keyboard event (WM_KEYUP or WM_KEYDOWN) to process.

The filter function must use the Pascal calling convention and must be declared **FAR**. The filter function must have the following form:

Filter Function

int FAR PASCAL *FilterFunc*(*nCode*, *wParam*, *lParam*)
int *nCode*;
WORD *wParam*;
DWORD *lParam*;

FilterFunc is a placeholder for the library-supplied function name. The actual name must be exported by including it in an **EXPORTS** statement in the library's module-definition file.

Parameter	Description
nCode	Specifies whether the filter function should process the message or call the **DefHookProc** function. If this value is HC_NOREMOVE, the application is using the **PeekMessage** function with the PM_NOREMOVE option and the message will not be removed from the system queue. If this value is less than zero, the filter function should pass the message to **DefHookProc** without further processing.
wParam	Specifies the virtual-key code of the given key.
lParam	Specifies the repeat count, scan code, key-transition code, previous key state, and context code, as shown in the following list. Bit 1 is the low-order bit:

Parameter	Description	
	Bit	**Value**
	0–15 (low-order word)	Repeat count (the number of times the keystroke is repeated as a result of the user holding down the key).
	16–23 (low byte of high-order word)	Scan code (OEM-dependent value).
	24	Extended key (1 if it is an extended key).
	25–26	Not used.
	27–28 (Context code (1 if the ALT key was held down while the key was pressed, 0 otherwise)	Used internally by Windows.
	30	Previous key state (1 if the key was held down before the message was sent, 0 if the key was up).
	31	Transition state (1 if the key is being released, 0 if the key is being pressed).

Return Value

The return value specifies what should happen to the message. It is zero if the message should be processed by Windows; it is 1 if the message should be discarded.

WH_MSGFILTER

Windows calls the WH_MSGFILTER filter function whenever a dialog box, message box, or menu has retrieved a message, and before it has processed that message. The filter allows an application to process or modify the messages.

NOTE This is the only task-specific filter. A task may install this filter.

The WH_MSGFILTER filter function must use the Pascal calling convention and must be declared **FAR**. The filter function must have the following form:

Filter Function

int FAR PASCAL *FilterFunc*(*nCode*, *wParam*, *lParam*)
int *nCode*;
WORD *wParam*;
DWORD *lParam*;

FilterFunc is a placeholder for the library- or application-supplied function name. The actual name must be exported by including it in an **EXPORTS** statement in the application's module-definition file.

Parameter	Description
nCode	Specifies the type of message being processed. It must be one of the following values:

Value	Meaning
MSGF_DIALOGBOX	Processing messages inside a dialog-box or message-box function.
MSGF_MENU	Processing keyboard and mouse messages in a menu.

If this value is less than zero, the filter function should pass this message to **DefHookProc** without further processing.

Parameter	Description
wParam	Specifies a NULL value.
lParam	Points to the message structure.

Return Value

The return value specifies the outcome of the function. It is nonzero if the hook function processes the message. Otherwise, it is zero.

WH_SYSMSGFILTER

Windows calls the WH_SYSMSGFILTER filter function whenever a dialog box, message box, or menu has retrieved a message and before it has processed that message. The filter allows an application to process or modify messages for any application in the system.

The filter function must use the Pascal calling convention and must be declared **FAR**. The filter function must have the following form:

Filter Function

int FAR PASCAL *FilterFunc*(*nCode*, *wParam*, *lParam*)
int *nCode*;
WORD *wParam*;
DWORD *lParam*;

FilterFunc is a placeholder for the library-supplied function name. The actual name must be exported by including it in an **EXPORTS** statement in the library's module-definition file.

Parameter	Description
nCode	Specifies the type of message being processed. It must be one of the following values:

Value	Meaning
MSGF_DIALOGBOX	Processing messages inside the **DialogBox** function.
MSGF_MENU	Processing keyboard and mouse messages in menu.
MSGF_MESSAGEBOX	Processing messages inside the **MessageBox** function.

If this value is less than zero, the filter function should pass the message to **DefHookProc** without further processing.

Parameter	Description
wParam	Specifies a NULL value.
lParam	Points to the message structure.

Return Value

The return value specifies the outcome of the function. It is nonzero if the hook function processes the message. Otherwise, it is zero.

SetWindowText

Syntax **void SetWindowText**(*hWnd*, *lpString*)

This function sets the given window's caption title (if one exists) to the string pointed to by the *lpString* parameter. If the *hWnd* parameter is a handle to a control, the **SetWindowText** function sets the text within the control instead of within the caption.

Parameter	Type/Description
hWnd	**HWND** Identifies the window or control whose text is to be changed.
lpString	**LPSTR** Points to a null-terminated character string.

Return Value None.

SetWindowWord

Syntax **WORD SetWindowWord**(*hWnd, nIndex, wNewWord*)

This function changes an attribute of the window specified by the *hWnd* parameter.

Parameter	Type/Description
hWnd	**HWND** Identifies the window to be modified.
nIndex	**int** Specifies the byte offset of the word to be changed. It can also be one of the following values:
wNewWord	**WORD** Specifies the replacement value.

Value	Meaning
GWW_HINSTANCE	Instance handle of the module that owns the window.
GWW_ID	Control ID of the child window.

Return Value The return value specifies the previous value of the specified word.

Comments To access any extra two-byte values allocated when the window-class structure was created, use a positive byte offset as the index specified by the *nIndex* parameter, starting at zero for the first two-byte value in the extra space, 2 for the next two-byte value and so on.

ShowCaret

Syntax

void ShowCaret(*hWnd*)

This function shows the caret on the display at the caret's current position. Once shown, the caret begins flashing automatically.

The **ShowCaret** function shows the caret only if it has a current shape and has not been hidden two or more times in a row. If the caret is not owned by the given window, the caret is not shown. If the *hWnd* parameter is NULL, the **SetCaret** function shows the caret only if it is owned by a window in the current task.

Hiding the caret is accumulative. If the **HideCaret** function has been called five times in a row, **ShowCaret** must be called five times to show the caret.

Parameter	Type/Description
hWnd	**HWND** Identifies the window that owns the caret, or is NULL to specify indirectly the owner window in the current task.

Return Value

None.

Comments

The caret is a shared resource. A window should show the caret only when it has the input focus or is active.

ShowCursor

Syntax

int ShowCursor(*bShow*)

This function shows or hides the cursor. The **ShowCursor** function actually sets an internal display counter that determines whether the cursor should be displayed. If the *bShow* parameter is nonzero, **ShowCursor** adds one to the display count. If *bShow* is zero, the display count is decreased by one. The cursor is displayed only if the display count is greater than or equal to zero. Initially, the display count is zero if a mouse is installed. Otherwise, it is –1.

Parameter	Type/Description
bShow	**BOOL** Specifies whether the display count is to be increased or decreased. The display count is increased if *bShow* is nonzero. Otherwise, it is decreased.

Return Value The return value specifies the new display count.

Comments The cursor is a shared resource. A window that hides the cursor should show the cursor before the cursor leaves its client area, or before the window relinquishes control to another window.

ShowOwnedPopups

Syntax void **ShowOwnedPopups**(*hWnd*,*fShow*)

This function shows or hides all pop-up windows that are associated with the *hWnd* parameter. If the *fShow* parameter is nonzero, all hidden pop-up windows are shown; if *fShow* is zero, all visible pop-up windows are hidden.

Parameter	Type/Description
hWnd	**HWND** Identifies the window that owns the pop-up windows that are to be shown or hidden.
fShow	**BOOL** Specifies whether or not pop-up windows are hidden. It is nonzero if all hidden pop-up windows should be shown; it is zero if all visible pop-up windows should be hidden.

Return Value None.

ShowScrollBar

Syntax void **ShowScrollBar**(*hWnd*, *wBar*, *bShow*)

This function displays or hides a scroll bar, depending on the value of the *bShow* parameter. If *bShow* is nonzero, the scroll bar is displayed; if *bShow* is zero, the scroll bar is hidden.

Parameter	Type/Description
hWnd	**HWND** Identifies a window that contains a scroll bar in its nonclient area if the *wBar* parameter is SB_HORZ, SB_VERT, or SB_BOTH. If *wBar* is SB_CTL, *hWnd* identifies a scroll-bar control.
wBar	**WORD** Specifies whether the scroll bar is a control or part of a window's nonclient area. If it is part of the nonclient area, *wBar* also indicates whether the scroll bar is positioned horizontally, vertically, or both. It must be one of the following values:

Parameter	Type/Description	
	Value	**Meaning**
	SB_BOTH	Specifies the window's horizontal and vertical scroll bars.
	SB_CTL	Specifies that the scroll bar is a control.
	SB_HORZ	Specifies the window's horizontal scroll bar.
	SB_VERT	Specifies the window's vertical scroll bar.
bShow	**BOOL** Specifies whether or not Windows hides the scroll bar. If *bShow* is zero, the scroll bar is hidden. Otherwise, the scroll bar is displayed.	

Return Value None.

Comments An application should not call this function to hide a scroll bar while processing a scroll-bar notification message.

ShowWindow

Syntax **BOOL ShowWindow**(*hWnd*, *nCmdShow*)

This function displays or removes the given window, as specified by the *nCmdShow* parameter.

Parameter	Type/Description
hWnd	**HWND** Identifies the window.
nCmdShow	**int** Specifies how the window is to be shown. It must be one of the values shown in Table 4.18, "Window States."

Return Value The return value specifies the previous state of the window. It is nonzero if the window was previously visible. It is zero if the window was previously hidden.

Comments The **ShowWindow** function must be called only once per program with the *nCmdShow* parameter from the WinMain function. Subsequent calls to **ShowWindow** must use one of the values listed above, instead of one specified by the *nCmdShow* parameter from the WinMain function. Table 4.18 lists the values for the *nCmdShow* parameter:

Table 4.18 Window States

Value	Meaning
SW_HIDE	Hides the window and passes activation to another window.
SW_MINIMIZE	Minimizes the specified window and activates the top-level window in the window-manager's list.
SW_RESTORE	Same as SW_SHOWNORMAL.
SW_SHOW	Activates a window and displays it in its current size and position.
SW_SHOWMAXIMIZED	Activates the window and displays it as a maximized window.
SW_SHOWMINIMIZED	Activates the window and displays it as iconic.
SW_SHOWMINNOACTIVE	Displays the window as iconic. The window that is currently active remains active.
SW_SHOWNA	Displays the window in its current state. The window that is currently active remains active.
SW_SHOWNOACTIVATE	Displays a window in its most recent size and position. The window that is currently active remains active.
SW_SHOWNORMAL	Activates and displays a window. If the window is minimized or maximized, Windows restores it to its original size and position.

SizeofResource

Syntax WORD **SizeofResource**(*hInstance*, *hResInfo*)

This function supplies the size (in bytes) of the specified resource. It is typically used with the **AccessResource** function to prepare memory to receive a resource from the file.

Parameter	Type/Description
hInstance	**HANDLE** Identifies the instance of the module whose executable file contains the resource.
hResInfo	**HANDLE** Identifies the desired resource. This handle is assumed to have been created by using the **FindResource** function.

Return Value The return value specifies the number of bytes in the resource. It is zero if the resource cannot be found.

Comments The value returned may be larger than the actual resource due to alignment. An application should not rely upon this value for the exact size of a resource.

StartSound

Syntax **int StartSound( )**

This function starts play in each voice queue. The **StartSound** function is not destructive, so it may be called any number of times to replay the current queues.

This function has no parameters.

Return Value Although the return-value type is integer, its contents should be ignored.

StopSound

Syntax **int StopSound( )**

This function stops playing all voice queues, then flushes the contents of the queues. The sound driver for each voice is turned off.

This function has no parameters.

Return Value Although the return-value type is integer, its contents should be ignored.

StretchBlt

Syntax **BOOL StretchBlt(***hDestDC, X, Y, nWidth, nHeight, hSrcDC, XSrc, YSrc, nSrcWidth, nSrcHeight, dwRop***)**

This function moves a bitmap from a source rectangle into a destination rectangle, stretching or compressing the bitmap if necessary to fit the dimensions of the destination rectangle. The **StretchBlt** function uses the stretching mode of the destination device context (set by the **SetStretchBltMode** function) to determine how to stretch or compress the bitmap.

StretchBlt moves the bitmap from the source device given by the *hSrcDC* parameter to the destination device given by the *hDestDC* parameter. The *XSrc*, *YSrc*, *nSrcWidth*, and *nSrcHeight* parameters define the origin and dimensions of the source rectangle. The *X*, *Y*, *nWidth*, and *nHeight* parameters give the origin and dimensions of the destination rectangle. The raster operation specified by the *dwRop* parameter defines how the source bitmap and the bits already on the destination device are combined.

StretchBlt creates a mirror image of a bitmap if the signs of the *nSrcWidth* and *nWidth* or *nSrcHeight* and *nHeight* parameters differ. If *nSrcWidth* and *nWidth* have different signs, the function creates a mirror image of the bitmap along the *x*-axis. If *nSrcHeight* and *nHeight* have different signs, the function creates a mirror image of the bitmap along the *y*-axis.

Parameter	Type/Description
hDestDC	**HDC** Identifies the device context to receive the bitmap.
X	**int** Specifies the logical *x*-coordinate of the upper-left corner of the destination rectangle.
Y	**int** Specifies the logical *y*-coordinate of the upper-left corner of the destination rectangle.
nWidth	**int** Specifies the width (in logical units) of the destination rectangle.
nHeight	**int** Specifies the height (in logical units) of the destination rectangle.
hSrcDC	**HDC** Identifies the device context that contains the source bitmap.
XSrc	**int** Specifies the logical *x*-coordinate of the upper-left corner of the source rectangle.
YSrc	**int** Specifies the logical *y*-coordinate of the upper-left corner of the source rectangle.
nSrcWidth	**int** Specifies the width (in logical units) of the source rectangle.
nSrcHeight	**int** Specifies the height (in logical units) of the source rectangle.
dwRop	**DWORD** Specifies the raster operation to be performed. Raster operation codes define how GDI combines colors in output operations that involve a current brush, a possible source bitmap, and a destination bitmap. For a list of raster-operation codes, see the **BitBlt** function, earlier in this chapter. For a complete list of the operations, see Chapter 11, "Binary and Ternary Raster-Operation Codes," in *Reference, Volume 2.*

Return Value

The return value specifies whether the bitmap is drawn. It is nonzero if the bitmap is drawn. Otherwise, it is zero.

Comments

StretchBlt stretches or compresses the source bitmap in memory, then copies the result to the destination. If a pattern is to be merged with the result, it is not merged until the stretched source bitmap is copied to the destination.

If a brush is used, it is the selected brush in the destination device context.

The destination coordinates are transformed according to the destination device context; the source coordinates are transformed according to the source device context.

If destination, source, and pattern bitmaps do not have the same color format, **StretchBlt** converts the source and pattern bitmaps to match the destination bitmaps. The foreground and background colors of the destination are used in the conversion.

If **StretchBlt** must convert a monochrome bitmap to color, it sets white bits (1) to background color and black bits (0) to foreground color. To convert color to monochrome, it sets pixels that match the background color to white (1), and sets all other pixels to black (0). The foreground and background colors of the device context with color are used.

Not all devices support the **StretchBlt** function. For more information, see the RC_BITBLT capability in the **GetDeviceCaps** function, earlier in this chapter.

StretchDIBits 3.0

Syntax

WORD StretchDIBits(*hDC*, *DestX*, *DestY*, *wDestWidth*, *wDestHeight*, *SrcX*, *SrcY*, *wSrcWidth*, *wSrcHeight*, *lpBits*, *lpBitsInfo*, *wUsage*, *dwRop*)

This function moves a device-independent bitmap (DIB) from a source rectangle into a destination rectangle, stretching or compressing the bitmap if necessary to fit the dimensions of the destination rectangle. The **StretchDIBits** function uses the stretching mode of the destination device context (set by the **SetStretchBltMode** function) to determine how to stretch or compress the bitmap.

StretchDIBits moves the bitmap from the device-independent bitmap specified by the *lpBits*, *lpBitsInfo*, and *wUsage* parameters to the destination device specified by the *hDC* parameter. The *XSrc*, *YSrc*, *wSrcWidth*, and *wSrcHeight* parameters define the origin and dimensions of the source rectangle. The origin of coordinate system of the device-independent bitmap is the lower-left corner. The *DestX*, *DestY*, *wDestWidth*, and *wDestHeight* parameters give the origin and dimensions of the destination rectangle. The origin of the coordinates of the destination depends on the current mapping mode of the device context. See the **SetMapMode** function earlier in this chapter for more information on mapping modes.

The raster operation specified by the *dwRop* parameter defines how the source bitmap and the bits already on the destination device are combined.

StretchDIBits creates a mirror image of a bitmap if the signs of the *wSrcWidth* and *wDestWidth* or *wSrcHeight* and *wDestHeight* parameters differ. If *wSrcWidth* and *nWidth* have different signs, the function creates a mirror image of the bitmap along the *x*-axis. If *wSrcHeight* and *nHeight* have different signs, the function creates a mirror image of the bitmap along the *y*-axis.

Parameter	Type/Description
hDC	**HDC** Identifies the destination device context for a display surface or memory bitmap.
DestX	**WORD** Specifies the *x*-coordinate (in logical units) of the origin of the destination rectangle.
DestY	**WORD** Specifies the *y*-coordinate (in logical units) of the origin of the destination rectangle.
wDestWidth	**WORD** Specifies the *x*-extent (in logical units) of the destination rectangle.
wDestHeight	**WORD** Specifies the *y*-extent (in logical units) of the destination rectangle.
SrcX	**WORD** Specifies the *x*-coordinate (in pixels) of the source in the DIB.
SrcY	**WORD** Specifies the *y*-coordinate (in pixels) of the source in the DIB.
wSrcWidth	**WORD** Specifies the width (in pixels) of the source rectangle in the DIB.
wSrcHeight	**WORD** Specifies the height (in pixels) of the source rectangle in the DIB.
lpBits	**LPSTR** Points to the DIB bits that are stored as an array of bytes.
lpBitsInfo	**LPBITMAPINFO** Points to a **BITMAPINFO** data structure that contains information about the DIB.
wUsage	**WORD** Specifies whether the **bmiColors[ ]** fields of the *lpBitsInfo* parameter contain explicit RGB values or indexes into the currently realized logical palette. The *wUsage* parameter must be one of the following values:

Value	Meaning
DIB_PAL_COLORS	The color table consists of an array of 16-bit indexes into the currently realized logical palette.
DIB_RGB_COLORS	The color table contains literal RGB values.

Parameter	Type/Description
dwRop	**DWORD** Specifies the raster operation to be performed. Raster operation codes define how GDI combines colors in output operations that involve a current brush, a possible source bitmap, and a destination bitmap. For a list of raster-operation codes, see the **BitBlt** function, earlier in this chapter. For a complete list of the operations, see Chapter 11, "Binary and Ternary Raster-Operation Codes," in *Reference, Volume 2*.

Return Value The return value is the number of scan lines copied.

Comments This function also accepts a device-independent bitmap specification formatted for Microsoft OS/2 Presentation Manager versions 1.1 and 1.2 if the *lpBitsInfo* parameter points to a **BITMAPCOREINFO** data structure.

SwapMouseButton

Syntax **BOOL SwapMouseButton**(*bSwap*)

This function reverses the meaning of left and right mouse buttons. If the *bSwap* parameter is TRUE, the left button generates right-button mouse messages and the right button generates left-button messages. If *bSwap* is FALSE, the buttons are restored to their original meaning.

Parameter	Type/Description
bSwap	**BOOL** Specifies whether the button meanings are reversed or restored.

Return Value The return value specifies the outcome of the function. It is TRUE if the fuction reversed the meaning of the mouse buttons. Otherwise, it is FALSE.

Comments Button swapping is provided as a convenience to people who use the mouse with their left hands. The **SwapMouseButton** function is usually called by the control panel only. Although applications are free to call the function, the mouse is a shared resource and reversing the meaning of the mouse button affects all applications.

SwapRecording 3.0

Syntax void **SwapRecording**(*wFlag*)

When running Microsoft Windows Swap, this function begins or ends analyzing swapping behavior. For more information on Swap, see *Tools*.

Parameter	Type/Description
wFlag	**WORD** Specifies whether Swap is to start or stop analyzing swapping behavior. The following are acceptable values:

Value	Meaning
0	Specifies that Swap stop analyzing.
1	Record swap calls, discard swap returns.
2	Same as 1, plus calls through thunks. This option records a large amount of data.

Return Value None.

SwitchStackBack 3.0

Syntax void **SwitchStackBack**( )

This function returns the stack of the current task to the task's data segment after it had been previously redirected by the **SwitchTasksBack** function.

This function has no parameters.

Return Value None.

Comments This function preserves the contents of the AX:DX register when it returns.

SwitchStackTo 3.0

Syntax void **SwitchStackTo**(*wStackSegment*, *wStackPointer*, *wStackTop*)

This function changes the stack of the current task to the segment identified by the *wStackSegment* parameter.

Dynamic-link libraries (DLLs) do not have a stack; instead, a DLL uses the stack of the task which calls the library. As a result, DLL functions that assume that the contents of the code-segment (CS) and stack-segment (SS) registers are the same will fail. The **Switch-StackTo** function redirects the stack of the task to the data segment of a DLL, permitting the DLL to call these functions. **SwitchStackTo** copies the arguments on the stack of the task to the new stack location.

Parameter	Type/Description
wStackSegment	**WORD** Specifies the data segment which is to contain the stack.
wStackPointer	**WORD** Specifies the offset of the beginning of the stack in the data segment.
wStackTop	**WORD** Specifies the offset of the top of the stack from the beginning of the stack.

Return Value None.

Comments A task can call **SwitchStackTo** before calling a function in a DLL that assumes the CS and DS registeres are equal. When the DLL function returns, the task must then call **Switch-StackBack** to redirect its stack to its own data segment.

A DLL can also call **SwitchStackTo** before calling a routine that assumes CS and DS are equal and then call **SwitchStackBack** before returning to the task that called the DLL function.

Calls to **SwitchStackTo** and **SwitchStackBack** cannot be nested. That is, after calling **SwitchStackTo**, a program must call **SwitchStackBack** before calling **SwitchStackTo** again.

SyncAllVoices

Syntax **int** **SyncAllVoices**( )

This function queues a sync mark in each queue. Upon encountering a sync mark in a voice queue, the voice is turned off until sync marks are encountered in all other queues. This forces synchronization among all voices.

This function has no parameters.

Return Value The return value specifies the result of the function. It is zero if the function is successful. If a voice queue is full, the return value is S_SERQFUL.

TabbedTextOut 3.0

Syntax

long TabbedTextOut(*hDC*, *X*, *Y lpString*, *nCount*, *nTabPositions*, *lpnTabStopPositions*, *nTabOrigin*)

This function writes a character string on the specified display, using the currently selected font and expanding tabs to the columns specified in the *lpnTabStopPositions* field.

Parameter	Type/Description
hDC	**HDC** Identifies the device context.
X	**int** Specifies the logical *x*-coordinate of the starting point of the string.
Y	**int** Specifies the logical *y*-coordinate of the starting point of the string.
lpString	**LPSTR** Points to the character string that is to be drawn.
nCount	**int** Specifies the number of characters in the string.
nTabPositions	**int** Specifies the number of tab-stop positions in the array to which the *lpnTabStopPositions* points.
lpnTabStopPositions	**LPINT** Points to an array of integers containing the tab-stop positions in pixels. The tab stops must be sorted in increasing order; back tabs are not allowed.
nTabOrigin	**int** Specifies the logical *x*-coordinate of the starting position from which tabs are expanded.

Return Value

The return value specifies the dimensions of the string. The height is in the high-order word; the width is in the low-order word.

Comments

If the *nTabPositions* parameter is zero the the *lpnTabStopPositions* parameter is NULL, tabs are expanded to eight average character widths.

If *nTabPositions* is 1, the tab stops will be separated by the distance specified by the first value in the array to which *lpnTabStopPositions* points.

If *lpnTabStopPositions* points to more than a single value, then a tab stop is set for each value in the array, up to the number specified by *nTabPositions*.

The *nTabOrigin* parameter allows an application to call the **TabbedTextOut** function several times for a single line. If the application calls **TabbedTextOut** more than once with the *nTabOrigin* set to the same value each time, the function expands all tabs relative to the position specified by *nTabOrigin*.

TextOut

Syntax

BOOL **TextOut**(*hDC*, *X*, *Y*, *lpString*, *nCount*)

This function writes a character string on the specified display, using the currently selected font. The starting position of the string is given by the *X* and *Y* parameters.

Parameter	Type/Description
hDC	**HDC** Identifies the device context.
X	**int** Specifies the logical *x*-coordinate of the starting point of the string.
Y	**int** Specifies the logical *y*-coordinate of the starting point of the string.
lpString	**LPSTR** Points to the character string that is to be drawn.
nCount	**int** Specifies the number of characters in the string.

Return Value

The return value specifies whether or not the string is drawn. It is nonzero if the string is drawn. Otherwise, it is zero.

Comments

Character origins are defined to be at the upper-left corner of the character cell.

By default, the current position is not used or updated by this function. However, an application can call the **SetTextAlign** function with the *wFlags* parameter set to TA_UPDATECP to permit Windows to use and update the current position each time the application calls **TextOut** for a given device context. When this flag is set, Windows ignores the *X* and *Y* parameters on subsequent **TextOut** calls.

Throw

Syntax

void **Throw**(*lpCatchBuf*, *nThrowBack*)

This function restores the execution environment to the values saved in the buffer pointed to by the *lpCatchBuf* parameter. The execution environment is the state of all system registers and the instruction counter. Execution continues at the **Catch** function that copied the environment pointed to by *lpCatchBuf*. The *nThrowBack* parameter is passed as the return value to the **Catch** function. It can be a nonzero value.

Parameter	Type/Description
lpCatchBuf	**LPCATCHBUF** Points to an array that contains the execution environment.
nThrowBack	**int** Specifies the value to be returned to the **Catch** function.

Return Value None.

Comments The **Throw** function is similar to the C run-time **LongJmp** function (which is incompatible with the Windows environment).

ToAscii 3.0

Syntax **int** **ToAscii**(*wVirtKey*, *wScanCode*, *lpKeyState*, *lpChar*, *wFlags*)

This function translates the virtual-key code specified by the *wVirtKey* parameter and the current keyboard state specified by the *lpKeyState* parameter to the corresponding ANSI character or characters.

Parameter	Type/Description
wVirtKey	**WORD** Specifies the virtual-key code to be translated.
wScanCode	**WORD** Specifies the "hardware" raw scan code of the key to be translated. The high-order bit of this value is set if the key is up.
lpKeyState	**LPSTR** Points to an array of 256 bytes, each of which contains the state of one key. If the high-order bit of the byte is set the key is up.
lpChar	**LPVOID** Points to a 32-bit buffer which receives the translated ANSI character or characters.
wFlags	**WORD** The bit 0 flag's menu display.

Return Value The return value specifies the number of characters copied to the buffer identified by the *lpChar* parameter. The return value is negative if the key was a dead key. Otherwise, it is one of the following values:

Value	Meaning
2	Two characters were copied to the buffer. This is usually an accent and a dead-key character, when the dead key cannot be translated otherwise.
1	One ANSI character was copied to the buffer.
0	The specified virtual key has no translation for the current state of the keyboard.

Comments The parameters supplied to the **ToAscii** function might not be sufficient to translate the virtual-key code because a previous dead key is stored in the keyboard driver.

Typically, **ToAscii** performs the translation based on the virtual-key code. In some cases, however, the *wScanCode* parameter may be used to distinguish between a key depression or a key release. The scan code is used for translating ALT+NUMBER key combinations.

TrackPopupMenu 3.0

Syntax **BOOL TrackPopupMenu**(*hMenu, wFlags, x, y, nReserved, hWnd, lpReserved*)

This function displays a "floating" pop-up menu at the specified location and tracks the selection of items on the pop-up menu. A floating pop-up menu can appear anywhere on the screen. The *hMenu* parameter specifies the handle of the menu to be displayed; the application obtains this handle by calling **CreatePopupMenu** to create a new pop-up menu or by calling **GetSubMenu** to retrieve the handle of a pop-up menu associated with an existing menu item.

Windows sends messages generated by the menu to the window identified by the *hWnd* parameter.

Parameter	Type/Description
hMenu	**HMENU** Identifies the pop-up menu to be displayed.
wFlags	**WORD** Not used. This parameter must be set to zero.
x	**int** Specifies the horizontal position in screen coordinates of the left side of the menu on the screen.
y	**int** Specifies the vertical position in screen coordinates of the top of the menu on the screen.
nReserved	**int** Is reserved and must be set to zero.

Parameter	Type/Description
hWnd	**HWND** Identifies the window which owns the pop-up menu. This window receives all WM_COMMAND messages from the menu.
lpReserved	**LPVOID** Is reserved and must be set to NULL.

Return Value The return value specifies the outcome of the function. It is TRUE if the function is successful. Otherwise, it is FALSE.

TranslateAccelerator

Syntax **int TranslateAccelerator**(*hWnd*, *hAccTable*, *lpMsg*)

This function processes keyboard accelerators for menu commands. The **TranslateAccelerator** function translates WM_KEYUP and WM_KEYDOWN messages to WM_COMMAND or WM_SYSCOMMAND messages, if there is an entry for the key in the application's accelerator table. The high-order word of the *lParam* parameter of the WM_COMMAND or WM_SYSCOMMAND message contains the value 1 to differentiate the message from messages sent by menus or controls.

WM_COMMAND or WM_SYSCOMMAND messages are sent directly to the window, rather than being posted to the application queue. The **TranslateAccelerator** function does not return until the message is processed.

Accelerator key strokes that are defined to select items from the system menu are translated into WM_SYSCOMMAND messages; all other accelerators are translated into WM_COMMAND messages.

Parameter	Type/Description
hWnd	**HWND** Identifies the window whose messages are to be translated.
hAccTable	**HANDLE** Identifies an accelerator table (loaded by using the **LoadAccelerators** function).
lpMsg	**LPMSG** Points to a message retrieved by using the **GetMessage** or **PeekMessage** function. The message must be an **MSG** data structure and contain message information from the Windows application queue.

Return Value The return value specifies the outcome of the function. It is nonzero if translation occurs. Otherwise, it is zero.

Comments

When **TranslateAccelerator** returns nonzero (meaning that the message is translated), the application should *not* process the message again by using the **TranslateMessage** function.

Commands in accelerator tables do not have to correspond to menu items.

If the accelerator command does correspond to a menu item, the application is sent WM_INITMENU and WM_INITMENUPOPUP messages, just as if the user were trying to display the menu. However, these messages are not sent if any of the following conditions are present:

- The window is disabled.
- The menu item is disabled.
- The command is not in the System menu and the window is minimized.
- A mouse capture is in effect (for more information, see the **SetCapture** function, earlier in this chapter).

If the window is the active window and there is no keyboard focus (generally true if the window is minimized), then WM_SYSKEYUP and WM_SYSKEYDOWN messages are translated instead of WM_KEYUP and WM_KEYDOWN messages.

If an accelerator key stroke that corresponds to a menu item occurs when the window that owns the menu is iconic, no WM_COMMAND message is sent. However, if an accelerator key stroke that does not match any of the items on the window's menu or the System menu occurs, a WM_COMMAND message is sent, even if the window is iconic.

TranslateMDISysAccel 3.0

Syntax

BOOL **TranslateMDISysAccel**(*hWndClient*, *lpMsg*)

This function processes keyboard accelerators for multiple document interface (MDI) child window System-menu commands. The **TranslateMDISysAccel** function translates WM_KEYUP and WM_KEYDOWN messages to WM_SYSCOMMAND messages. The high-order word of the *lParam* parameter of the WM_SYSCOMMAND message contains the value 1 to differentiate the message from messages sent by menus or controls.

Parameter	Type/Description
hWndClient	**HWND** Identifies the parent MDI client window.
lpMsg	**LPMSG** Points to a message retrieved by using the **GetMessage** or **PeekMessage** function. The message must be an **MSG** data structure and contain message information from the Windows application queue.

Return Value The return value is TRUE if the function translated a message into a system command. Otherwise, it is FALSE.

TranslateMessage

Syntax **BOOL TranslateMessage**(*lpMsg*)

This function translates virtual-key messages into character messages, as follows:

- WM_KEYDOWN/WM_KEYUP combinations produce a WM_CHAR or a WM_DEADCHAR message.
- WM_SYSKEYDOWN/WM_SYSKEYUP combinations produce a WM_SYSCHAR or a WM_SYSDEADCHAR message.

The character messages are posted to the application queue, to be read the next time the application calls the **GetMessage** or **PeekMessage** function.

Parameter	Type/Description
lpMsg	**LPMSG** Points to an **MSG** data structure retrieved through the **GetMessage** or **PeekMessage** function. The structure contains message information from the Windows application queue.

Return Value The return value specifies the outcome of the function. It is nonzero if the message is translated (that is, character messages are posted to the application queue). Otherwise, it is zero.

Comments The **TranslateMessage** function does not modify the message given by the *lpMsg* parameter.

TranslateMessage produces WM_CHAR messages only for keys which are mapped to ASCII characters by the keyboard driver.

An application should not call **TranslateMessage** if the application processes virtual-key messages for some other purpose. For instance, an application should not call the **TranslateMessage** function if the **TranslateAccelerator** function returns nonzero.

TransmitCommChar

Syntax **int TransmitCommChar**(*nCid*, *cChar*)

This function marks the character specified by the *cChar* parameter for immediate transmission, by placing it at the head of the transmit queue.

Parameter	Type/Description
nCid	**int** Specifies the communication device to receive the character. The **OpenComm** function returns this value.
cChar	**char** Specifies the character to be transmitted.

Return Value

The return value specifies the result of the function. It is zero if the function is successful. It is negative if the character cannot be transmitted. A character cannot be transmitted if the character specified by the previous **TransmitCommChar** function has not yet been sent.

UngetCommChar

Syntax **int UngetCommChar**(*nCid, cChar*)

This function places the character specified by the *cChar* parameter back into the receive queue, making this character the first to be read on a subsequent read from the queue.

Consecutive calls to the **UngetCommChar** function are not allowed. The character placed back into the queue must be read before attempting to place another.

Parameter	Type/Description
nCid	**int** Specifies the communication device to receive the character.
cChar	**char** Specifies the character to be placed in the receive queue.

Return Value The return value specifies the outcome of the function. It is zero if the function is successful. It is negative if an error occurs.

UnhookWindowsHook

Syntax **BOOL UnhookWindowsHook**(*nHook, lpfnHook*)

This function removes the Windows hook function pointed to by the *lpfnHook* parameter from a chain of hook functions. A Windows hook function processes events before they are sent to an application's message loop in the WinMain function.

Parameter	Type/Description
nHook	**int** Specifies the type of hook function removed. It may be one of the following values:

Value	Meaning
WH_CALLWNDPROC	Installs a window-function filter.
WH_GETMESSAGE	Installs a message filter.
WH_JOURNALPLAYBACK	Installs a journaling playback filter.
WH_JOURNALRECORD	Installs a journaling record filter.
WH_KEYBOARD	Install a keyboard filter.

Parameter	Type/Description	
	Value	**Meaning**
	WH_MSGFILTER	Installs a message filter.
lpfnHook	**FARPROC** Is the procedure-instance address of the hook function.	

Return Value The return value specifies the outcome of the function. It is nonzero if the hook function is successfully removed. Otherwise, it is zero.

UnionRect

Syntax **int UnionRect**(*lpDestRect*, *lpSrc1Rect*, *lpSrc2Rect*)

This function creates the union of two rectangles. The union is the smallest rectangle that contains both source rectangles.

Parameter	Type/Description
lpDestRect	**LPRECT** Points to the **RECT** data structure that is to receive the new union.
lpSrc1Rect	**LPRECT** Points to a **RECT** data structure that contains a source rectangle.
lpSrc2Rect	**LPRECT** Points to a **RECT** data structure that contains a source rectangle.

Return Value The return value specifies the outcome of the function. It is nonzero if the union is not empty. It is zero if the union is empty.

Comments Windows ignores the dimensions of an "empty" rectangle, that is, a rectangle that has no height or has no width.

UnlockData

Syntax **HANDLE UnlockData**(*Dummy*)

This macro unlocks the current data segment. It is intended to be used by modules that have moveable data segments.

Parameter	Type/Description
Dummy	**int** Is not used; can be set to zero.

Return Value None.

UnlockResource

Syntax BOOL **UnlockResource**(*hResData*)

This macro unlocks the resource specified by the *hResData* parameter and decreases the resource's reference count by one.

Parameter	Type/Description
hResData	**HANDLE** Identifies the global memory block to be unlocked.

Return Value The return value specifies the outcome of the macro. It is zero if the block's reference count is decreased to zero. Otherwise, it is nonzero.

UnlockSegment

Syntax BOOL **UnlockSegment**(*wSegment*)

This function unlocks the segment whose segment address is specified by the *wSegment* parameter. If *wSegment* is –1, the **UnlockSegment** function unlocks the current data segment.

In real mode, or if the segment is discardable, **UnlockSegment** decreases the segment's lock count by one. In protected mode, **UnlockSegment** decreases the lock count of discardable objects and automatic data segments only. The segment is completely unlocked and subject to moving or discarding if the lock count is decreased to zero. Other functions also can affect the lock count of a memory object. See the description of the **GlobalFlags** function for a list of the functions that affect the lock count.

In all cases, each time an application calls **LockSegment** for a segment, it must eventually call **UnlockSegment** for the segment.

Parameter	Type/Description
wSegment	**WORD** Specifies the segment address of the segment to be unlocked. If *wSegment* is –1, the **UnlockSegment** function unlocks the current data segment.

Return Value The return value specifies the outcome of the function. It is zero if the segment's lock count was decreased to zero. Otherwise, the return value is nonzero. An application should not rely on the return value to determine the number of times it must subsequently call **UnlockSegment** for the segment.

UnrealizeObject

Syntax **BOOL UnrealizeObject**(*hObject*)

If the *hObject* parameter specifies a brush, this function directs GDI to reset the origin of the given brush the next time it is selected.

If *hObject* specifies a logical palette, this function directs GDI to realize the palette as though it had not previously been realized. The next time the application calls the **RealizePalette** function for the specified palette, GDI completely remaps the logical palette to the system palette.

Parameter	Type/Description
hObject	**HANDLE** Identifies the object to be reset.

Return Value The return value specifies the outcome of the function. It is nonzero if the function is successful. Otherwise, it is zero.

Comments The **UnrealizeObject** function should not be used with stock objects.

This function must be called whenever a new brush origin is set (by means of the **SetBrushOrigin** function).

A brush specified by the *hObject* parameter must not be the currently selected brush of any display context.

A palette specified by *hObject* can be the currently selected palette of a display context.

UnregisterClass 3.0

Syntax **BOOL UnregisterClass**(*lpClassName*, *hInstance*)

This function removes the window class specified by *lpClassName* from the window-class table, freeing the storage required for the class.

Parameter	Type/Description
lpClassName	**LPSTR** Points to a null-terminated string containing the class name. This class name must have been previously registered by calling the **RegisterClass** function with a valid **hInstance** field in the **WNDCLASS** structure parameter. Predefined classes, such as dialog-box controls, may not be unregistered.
hInstance	**HANDLE** Identifies the instance of the module that created the class.

Return Value The return value is TRUE if the function successfully removed the window class from the window-class table. It is FALSE if the class could not be found or if a window exists that was created with the class.

Comments Before using this function, destroy all windows created with the specified class.

UpdateColors 3.0

Syntax **int UpdateColors**(*hDC*)

This function updates the client area of the device context identified by the *hDC* parameter by matching the current colors in the client area to the system palette on a pixel-by-pixel basis. An inactive window with a realized logical palette may call **UpdateColors** as an alternative to redrawing its client area when the system palette changes. For more information on using color palettes, see *Guide to Programming*.

Parameter	Type/Description
hDC	**HDC** Identifies the device context.

Return Value The return value is not used.

Comments **UpdateColors** typically updates a client area faster than redrawing the area. However, because **UpdateColors** performs the color translation based on the color of each pixel before the system palette changed, each call to this function results in the loss of some color accuracy.

UpdateWindow

Syntax **void UpdateWindow**(*hWnd*)

This function updates the client area of the given window by sending a WM_PAINT message to the window if the update region for the window is not empty. The **UpdateWindow** function sends a WM_PAINT message directly to the window function of the given window, bypassing the application queue. If the update region is empty, no message is sent.

Parameter	Type/Description
hWnd	**HWND** Identifies the window to be updated.

Return Value None.

ValidateCodeSegments 3.0

Syntax void ValidateCodeSegments( )

This function outputs debugging information to a terminal if any code segments have been altered by random memory overwrites. It is only available in the debugging version of Windows and is enabled by default. To disable the function, set the **EnableSegmentChecksum** flag in the [kernel] section of WIN.INI to 0. Windows does not validate code segments in protected (standard or 386 enhanced) mode.

This function has no parameters.

Return Value None.

ValidateFreeSpaces

Syntax LPSTR ValidateFreeSpaces( )

This function (available only in the debugging version of Windows) checks free segments in memory for valid contents. In the debugging version of Windows, the kernel fills all the bytes in free segments with the hexadecimal value CC. This function begins checking for valid contents in the free segment with the lowest address, and continues checking until it finds an invalid byte or until it has determined that all free space contains valid contents. Before calling this function, put the following lines in the WIN.INI file:

```
[kernel]
EnableFreeChecking=1
EnableHeapChecking=1
```

This function has no parameters.

Return Value None.

Comments Windows sends debugging information to the debugging terminal if an invalid byte is encountered and performs a fatal exit.

The [kernel] entries in WIN.INI will cause automatic checking of free memory. Before returning a memory block to the application in response to a **GlobalAlloc** call, Windows checks that memory to make sure it is filled with 0CCH. Before a **GlobalCompact** call, all free memory is checked. Note that using this function slows Windows down system-wide by about 20%.

ValidateRect

Syntax void **ValidateRect**(*hWnd*, *lpRect*)

This function validates the client area within the given rectangle by removing the rectangle from the update region of the given window. If the *lpRect* parameter is NULL, the entire window is validated.

Parameter	Type/Description
hWnd	**HWND** Identifies the window whose update region is to be modified.
lpRect	**LPRECT** Points to a **RECT** data structure that contains the rectangle (in client coordinates) to be removed from the update region.

Return Value None.

Comments The **BeginPaint** function automatically validates the entire client area. Neither the **ValidateRect** nor **ValidateRgn** function should be called if a portion of the update region needs to be validated before the next WM_PAINT message is generated.

Windows continues to generate WM_PAINT messages until the current update region is validated.

ValidateRgn

Syntax void **ValidateRgn**(*hWnd*, *hRgn*)

This function validates the client area within the given region by removing the region from the current update region of the given window. If the *hRgn* parameter is NULL, the entire window is validated.

Parameter	Type/Description
hWnd	**HWND** Identifies the window whose update region is to be modified.
hRgn	**HRGN** Identifies a region that defines the area to be removed from the update region.

Return Value None.

Comments

The given region must have been created previously by means of a region function (for more information, see Chapter 1, "Window Manager Interface Functions"). The region coordinates are assumed to be client coordinates.

VkKeyScan

Syntax

int VkKeyScan (*cChar*)

This function translates an ANSI character to the corresponding virtual-key code and shift state for the current keyboard. Applications which send character by means of WM_KEYUP and WM_KEYDOWN messages use this function.

Parameter	Type/Description
cChar	**char** Specifies the character for which the corresponding virtual-key code is to be found.

Return Value

The VK_ code is returned in the low-order byte and the shift state in the high-order byte. The shift states are:

Value	Meaning
0	No shift.
1	Character is shifted.
2	Character is control character.
6	Charcter is CONTROL+ALT.
7	Character is SHIFT+CONTROL+ALT.
3, 4, 5	A shift key combination that is not used for characters.

If no key is found that translates to the passed ANSI code, a –1 is returned in both the low-order and high-order bytes.

Comments

Translations for the numeric keypad (VK_NUMPAD0 through VK_DIVIDE) are ignored. This function is intended to force a translation for the main keyboard only.

WaitMessage

Syntax void **WaitMessage**( )

This function yields control to other applications when an application has no other tasks to perform. The **WaitMessage** function suspends the application and does not return until a new message is placed in the application's queue.

This function has no parameters.

Return Value None.

Comments The **GetMessage**, **PeekMessage**, and **WaitMessage** functions yield control to other applications. These calls are the only way to let other applications run. If your application does not call any of these functions for long periods of time, other applications cannot run.

When **GetMessage**, **PeekMessage**, and **WaitMessage** yield control to other applications, the stack and data segments of the application calling the function may move in memory to accommodate the changing memory requirements of other applications. If the application has stored long pointers to objects in the data or stack segment (that is, global or local variables), these pointers can become invalid after a call to **GetMessage**, **PeekMessage**, or **WaitMessage**.

WaitSoundState

Syntax int **WaitSoundState**(*nState*)

This function waits until the play driver enters the specified state.

Parameter	Type/Description
nState	**int** Specifies the state of the voice queues. It can be any one of the following values:

Value	Meaning
S_ALLTHRESHOLD	All voices have reached threshold.
S_QUEUEEMPTY	All voice queues are empty and sound drivers turned off.
S_THRESHOLD	A voice queue has reached threshold, and returns voice.

Return Value The return value specifies the result of the function. It is zero if the function is successful. If the state is not valid, the return value is S_SERDST.

WindowFromPoint

Syntax **HWND WindowFromPoint**(*Point*)

This function identifies the window that contains the given point; *Point* must specify the screen coordinates of a point on the screen.

Parameter	Type/Description
Point	**POINT** Specifies a **POINT** data structure that defines the point to be checked.

Return Value The return value identifies the window in which the point lies. It is NULL if no window exists at the given point.

WinExec 3.0

Syntax **WORD WinExec**(*lpCmdLine*, *nCmdShow*)

This function executes the Windows or non-Windows application identified by the *lpCmdLine* parameter. The *nCmdShow* parameter specifies the initial state of the application's main window when it is created.

Parameter	Type/Description
lpCmdLine	**LPSTR** Points to a null-terminated character string that contains the command line (filename plus optional parameters) for the application to be executed. If the *lpCmdLine* string does not contain a directory path, Windows will search for the executable file in this order:

1. The current directory
2. The Windows directory (the directory containing WIN.COM); the **GetWindowsDirectory** function obtains the pathname of this directory
3. The Windows system directory (the directory containing such system files as KERNEL.EXE); the **GetSystemDirectory** function obtains the pathname of this directory

Parameter	Type/Description
	4. The directories listed in the PATH environment variable
	5. The list of directories mapped in a network
	If the application filename does not contain an extension, then .EXE is assumed.
nCmdShow	**int** Specifies how a Windows application window is to be shown. See the description of the **ShowWindow** function for a list of the acceptable values for the *nCmdShow* parameter. For a non-Windows application, the PIF file, if any, for the application determines the window state.

Return Value

The return value specifies whether the function was successful. If the function was successful, the return value is greater than 32. Otherwise, it is a value less than 32 that specifies the error. The following list describes the error values returned by this function:

Value	Meaning
0	Out of memory.
2	File not found.
3	Path not found.
5	Attempt to dynamically link to a task.
6	Library requires separate data segments for each task.
10	Incorrect Windows version.
11	Invalid .EXE file (non-Windows .EXE or error in .EXE image).
12	OS/2 application.
13	DOS 4.0 application.
14	Unknown .EXE type.
15	Attempt in protected (standard or 386 enhanced) mode to load an .EXE created for an earlier version of Windows.
16	Attempt to load a second instance of an .EXE containing multiple, writeable data segments.
17	Attempt in large-frame EMS mode to load a second instance of an application that links to certain nonshareable DLLs already in use.

Value	Meaning
18	Attempt in real mode to load an application marked for protected mode only.

Comments The **LoadModule** function provides an alternative method for executing a program.

WinHelp 3.0

Syntax **BOOL WinHelp**(*hWnd, lpHelpFile, wCommand, dwData*)

This function invokes the Windows Help application and passes optional data indicating the nature of the help requested by the application. The application specifies the name and, where required, the directory path of the help file which the Help application is to display. See *Tools* for information on creating and using help files.

Parameter	Type/Description
hWnd	**HWND** Identifies the window requesting help.
lpHelpFile	**LPSTR** Points to a null-terminated string containing the directory path, if needed, and the name of the help file which the Help application is to display.
wCommand	**WORD** Specifies the type of help requested. It may be any one of the following values:

Value	Meaning
HELP_CONTEXT	Displays help for a particular context identified by a 32-bit unsigned integer value in *dwData*.
HELP_HELPONHELP	Displays help for using the help application itself. If the *wCommand* parameter is set to HELP_HELPONHELP, **WinHelp** ignores the *lpHelpFile* and *dwData* parameters.
HELP_INDEX	Displays the index of the specified help file. An application should use this value only for help files with a single index. It should not use this value with HELP_SETINDEX.

Parameter	Type/Description	
	Value	**Meaning**
	HELP_KEY	Displays help for a particular key word identified by a string pointed to by *dwData*.
	HELP_MULTIKEY	Displays help for a key word in an alternate keyword table.
	HELP_QUIT	Notifies the help application that the specified help file is no longer in use.
	HELP_SETINDEX	Sets the context specified by the *dwData* parameter as the current index for the help file specified by the *lpHelpFile* parameter. This index remains current until the user accesses a different help file. To help ensure that the correct index remains set, the application should call **WinHelp** with *wCommand* set to HELP_SETINDEX (with *dwData* specifying the corresponding context identifier) following each call to **WinHelp** with *wCommand* set to HELP_CONTEXT. An application should use this value only for help files with more than one index. It should not use this value with HELP_INDEX.
dwData	**DWORD** Specifies the context or key word of the help requested. If *wCommand* is HELP_CONTEXT, *dwData* is a 32-bit unsigned integer containing a context-identifier number. If *wCommand* is HELP_KEY, *dwData* is a long pointer to a null-terminated string that contains a key word identifying the help topic. If *wCommand* is HELP_MULTIKEY, *dwData* is a long pointer to a **MULTIKEYHELP** data structure. Otherwise, *dwData* is ignored and should be set to NULL.	

Return Value The return value specifies the outcome of the function. It is TRUE if the function was successful. Otherwise it is FALSE.

Comments The application must call **WinHelp** with *wCommand* set to HELP_QUIT before closing the window that requested the help. The Help application will not actually terminate until all applications that have requested help have called **WinHelp** with *wCommand* set to HELP_QUIT.

WriteComm

Syntax **int WriteComm**(*nCid, lpBuf, nSize*)

This function writes the number of characters specified by the *nSize* parameter to the communication device specified by the *nCid* parameter from the buffer pointed to by the *lpBuf* parameter.

Parameter	Type/Description
nCid	**int** Specifies the device to receive the characters. The **OpenComm** function returns this value.
lpBuf	**LPSTR** Points to the buffer that contains the characters to be written.
nSize	**int** Specifies the number of characters to write.

Return Value The return value specifies the number of characters actually written. When an error occurs, the return value is set to a value less than zero, making the absolute value of the return value the actual number of characters written. The cause of the error can be determined by using the **GetCommError** function to retrieve the error code and status.

Comments The **WriteComm** function will delete data in the transmit queue if there is not enough room in the queue for the additional characters. Applications should check the available space in the transmit queue with the **GetCommError** function before calling **WriteComm**. Also, applications should use the **OpenComm** function to set the size of the transmit queue to an amount no smaller than the size of the largest expected output string.

WritePrivateProfileString 3.0

Syntax **BOOL WritePrivateProfileString**(*lpApplicationName, lpKeyName, lpString, lpFileName*)

This function copies the character string pointed to by the *lpString* parameter into the specified initialization file. It searches the file for the key named by the *lpKeyName* parameter under the application heading specified by the *lpApplicationName* parameter. If there is no match, it adds to the user profile a new string entry containing the key name and the key value specified by the *lpString* parameter. If there is a matching key, the function replaces that key's value with *lpString*.

Parameter	Type/Description
lpApplicationName	**LPSTR** Points to an application heading in the initialization file.
lpKeyName	**LPSTR** Points to a key name that appears under the application heading in the initialization file.
lpString	**LPSTR** Points to the string that contains the new key value.
lpFileName	**LPSTR** Points to a null-terminated character string that names the initialization file. If *lpFileName* does not contain a fully qualified pathname for the file, this function searches the Windows directory for the file. If the file does not exist and *lpFileName* does not contain a fully qualified pathname, this function creates the file in the Windows directory. The **WritePrivateProfileString** does not create a file if *lpFileName* contains the fully qualified pathname of a file that does not exist.

Return Value

The return value specifies the result of the function. It is nonzero if the function is successful. Otherwise, it is zero.

Comments

An application should use a private (application-specific) initialization file to record information which affects only that application. This improves both the performance of the application and Windows itself by reducing the amount of information that Windows must read when it accesses the initialization file.

If there is no application field for *lpApplicationName*, this function creates a new application field and places an appropriate key-value line in that field of the initialization file.

A string entry in the initialization file has the following form:

```
[application name]
keyname = string
      .
      .
      .
```

An application can also call **WritePrivateProfileString** to delete lines from its private initialization file. If *lpString* is NULL, the function deletes the entire line identified by the *lpKeyName* parameter. If *lpString* points to a null string, the function deletes only the value; the key name remains in the file. If *lpKeyName* is NULL, the function deletes the entire section identified by the *lpApplicationName* parameter; however, the function does not delete any lines beginning with the semicolon (;) comment character.

WriteProfileString

Syntax

BOOL WriteProfileString(*lpApplicationName*, *lpKeyName*, *lpString*)

This function copies the character string pointed to by the *lpString* parameter into the Windows initialization file, WIN.INI. It searches WIN.INI for the key named by the *lpKeyName* parameter under the application heading specified by the *lpApplicationName* parameter. If there is no match, it adds to the user profile a new string entry containing the key name and the key value specified by the *lpString* parameter. If there is a matching key, the function replaces that key's value with *lpString*.

Parameter	Type/Description
lpApplicationName	**LPSTR** Points to an application heading in WIN.INI.
lpKeyName	**LPSTR** Points to a key name that appears under the application heading WIN.INI.
lpString	**LPSTR** Points to the string that contains the new key value.

Return Value

The return value specifies the result of the function. It is nonzero if the function is successful. Otherwise, it is zero.

Comments

If there is no match for *lpApplicationName*, this function creates a new application field and adds the string pointed to by *lpString*.

A string entry in WIN.INI has the following form:

```
[application name]
keyname = string
.
.
.
```

An application can also call **WriteProfileString** to delete lines from WIN.INI. If *lpString* is NULL, the function deletes the entire line identified by the *lpKeyName* parameter. If *lpString* points to a null string, the function deletes only the value; the key name remains in the file. If *lpKeyName* is NULL, the function deletes the entire section identified by the *lpApplicationName* parameter; however, the function does not delete any lines beginning with the semicolon (;) comment character.

wsprintf 3.0

Syntax **int** **wsprintf**(*lpOutput*, *lpFormat*[[, *argument*]]. . .)

This function formats and stores a series of characters and values in a buffer. Each argument (if any) is converted and output according to the corresponding format specification in the format string. The function appends a NULL to the end of the characters written, but the return value does not include the terminating null character in its character count.

Parameter	Type/Description
lpOutput	**LPSTR** Points to a null-terminated character string to receive the formatted output.
lpFormat	**LPSTR** Points to a null-terminated character string that contains the format-control string. In addition to ordinary ASCII characters, a format specification for each argument appears in this string. See the following "Comments" section for more information on the format specification.
argument	Is one or more optional arguments. The number and type of *argument* parameters depends on the corresponding format-control character sequences in *lpFormat*.

Return Value The return value is the number of characters stored in *lpOutput*, not counting the terminating NULL. If an error occurs, the function returns a value less than the length of *lpFormat*.

Comments The format-control string contains format specifications that determine the output format for the arguments which follow the *lpFormat* parameter. Format specifications, discussed below, always begin with a percent sign (%). If a percent sign is followed by a character that has no meaning, such as a format field, the character is output as is. For example, *%%* produces a single percent-sign character.

The format-control string is read from left to right. When the first format specification (if any) is encountered, it causes the value of the first argument after the format-control string to be converted and output according to the format specification. The second format specification causes the second argument to be converted and output, and so on. If there are more arguments than there are format specifications, the extra arguments are ignored. The results are undefined if there are not enough arguments for all of the format specifications.

A format specification has the following form:

%[[**–**]][[**#**]][[**0**]][[*width*]][[*.precision*]]*type*

Each field of the format specification is a single character or a number signifying a particular format option. The *type* characters, which appear after the last optional format field, determine whether the associated argument is interpreted as a character, a string, or a number. The simplest format specification contains only the percent sign and a type character (for example, %s). The optional fields control other aspects of the formatting. The following shows the optional and required fields and their meaning:

Field	Description
–	Pad the output with blanks or zeroes to the right to fill the field width, justifying the output to the left. If this field is omitted, the output is padded to the left, justifying the output to the right.
#	Prefix hexadecimal values with 0x (lowercase) or OX (uppercase).
0	Pad the output value with zeroes to fill the field width. If this field is omitted, the output value is padded with blank spaces.
width	Output the specified minimum number of characters. The *width* field is a nonnegative integer. The width specification never causes a value to be truncated; if the number of characters in the output value is greater than the specified width, or if the *width* field is not present, all characters of the value are printed, subject to the precision specification.
precision	Output the specified minimum number of digits. If the number of digits in the argument is less than the specified precision, the output value is padded on the left with zeroes. The value is not truncated when the number of digits exceeds the specified precision. If the specified precision is 0, omitted entirely, or if the period (.) appears without a number following it, the precision is set to 1. For strings, output the specified maximum number of characters.
type	Output the corresponding argument as a character, string, or a number. This field may be any of the following character sequences:

Sequence	Meaning
s	Insert a string argument referenced by a long pointer. The argument corresponding to this sequence *must* be passed as a long pointer (**LPSTR**).
c	Insert a single character argument.
d, i	Insert a signed decimal integer argument.
ld, li	Insert a long signed decimal integer argument.
u	Insert an unsigned integer argument.
lu	Insert a long unsigned integer argument.

Field	Description	
	Sequence	**Meaning**
	x, X	Insert an unsigned hexadecimal integer argument in lowercase or uppercase.
	lx, lX	Insert a long unsigned hexadecimal integer argument in lowercase or uppercase.

NOTE Unlike all other Windows functions, **wsprintf** uses the C calling convention (**cdecl**), rather than the Pascal calling convention. As a result, it is the caller's responsibility to pop arguments off the stack, and arguments are pushed in reverse order (that is, the *lpOutput* parameter is pushed last, to the lowest address). In C-language modules, the C compiler performs this task.

wvsprintf 3.0

Syntax **int wvsprintf**(*lpOutput*, *lpFormat*, *lpArglist*)

This function formats and stores a series of characters and values in a buffer. The items pointed to by the argument list are converted and output according to the corresponding format specification in the format string. The function appends a NULL to the end of the characters written, but the return value does not include the terminating null character in its character count.

Parameter	Type/Description
lpOutput	**LPSTR** Points to a null-terminated character string to receive the formatted output.
lpFormat	**LPSTR** Points to a null-terminated character string that contains the format-control string. In addition to ordinary ASCII characters, a format specification for each argument appears in this string. See the description of the **wsprintf** function, earlier in this chapter, for more information on the format specification.

Parameter	Type/Description
lpArglist	**LPSTR** Points to an array of words, each of which specifies an arguement for the format-control string. The number, type and interpretation of the arguments depend on the corresponding format-control character sequences in *lpFormat*. Each character or word-sized integer (%c, %d, %x, %i) requires one word in *lpArglist*. Long integers (%ld, %li, %lx) require two words, the low-order word of the integer followed by the high-order word. A string (%s) requires two words, the offset followed by the segment (which together make up a far pointer).

Return Value

The return value is the number of characters stored in *lpOutput*, not counting the terminating NULL. If an error occurs, the function returns a value less than the length of *lpFormat*.

Yield

Syntax **void Yield( )**

This function halts the current task and starts any waiting task.

This function has no parameters.

Return Value None.

Comments Applications that contain windows should use a **DispatchMessage, PeekMessage**, or **TranslateMessage** loop rather than calling the **Yield** function directly. The **PeekMessage** loop handles message synchronization properly and yields at the appropriate times.

Part 2 Windows Messages

Part 2 provides reference information on Windows messages. Windows messages allow Windows applications to communicate with each other and with the Windows system within a nonpreemptive multitasking environment.

CHAPTERS

Chapter 5 Messages Overview

This chapter describes groups of related Microsoft Windows messages. Each section states the purpose of the message group, lists the messages in the group, and describes each message. Some of the sections contain additional information. See Chapter 1, "Window Manager Interface Functions," for an explanation of sending and receiving messages.

This chapter lists the following categories of Windows messages:

- Window-management messages
- Initialization messages
- Input messages
- System messages
- Clipboard messages
- System-information messages
- Control messages
- Notification messages
- Scroll-bar messages
- Nonclient-area messages
- Multiple document interface messages

5.1 Window-Management Messages

Window-management messages are sent by Windows to an application when the state of a window changes. The following list briefly describes each window-management message:

Message	Description
WM_ACTIVATE	Sent when a window becomes active or inactive.
WM_ACTIVATEAPP	Sent when the window being activated belongs to a different application than the window that was previously active.
WM_CANCELMODE	Cancels any mode the system is in, such as one that tracks the mouse in a scroll bar or moves a window. Windows sends the WM_CANCELMODE message when an application displays a message box.
WM_CHILDACTIVATE	Notifies a child window's parent window when the **SetWindowPos** function moves a child window.
WM_CLOSE	Sent whenever the window is closed.
WM_CREATE	Sent when the **CreateWindow** function is called.
WM_CTLCOLOR	Sent to the parent window of a predefined control or message box when the control or message box is about to be drawn.
WM_DESTROY	Sent when the **DestroyWindow** function is called, after the window has been removed from the screen.
WM_ENABLE	Sent after a window has been enabled or disabled.
WM_ENDSESSION	Tells an application that has responded nonzero to a WM_QUERYENDSESSION message whether the session is actually being ended.
WM_ENTERIDLE	Informs a window that a dialog box or menu is displayed and waiting for user action.
WM_ERASEBKGND	Sent when the window background needs to be erased.
WM_GETDLGCODE	Sent to an input procedure associated with a control.

Message	Description
WM_GETMINMAXINFO	Retrieves the maximized size of the window, the minimum or maximum tracking size of the window, and the maximized position of the window.
WM_GETTEXT	Copies the text that corresponds to a window.
WM_GETTEXTLENGTH	Retrieves the length (in bytes) of the text associated with a window.
WM_ICONERASEBKGND	Sent to an iconic window with a class icon when the background of the icon needs to be erased.
WM_KILLFOCUS	Sent immediately before a window loses the input focus.
WM_MENUCHAR	Notifies the window that owns the menu when the user presses a menu mnemonic character that doesn't match any of the predefined mnemonics in the current menu.
WM_MENUSELECT	Notifies a window that the user has selected a menu item.
WM_MOVE	Sent when a window is moved.
WM_PAINT	Sent whenever Windows or an application makes a request to repaint a portion of an application's window.
WM_PAINTICON	Sent whenever Windows or an application makes a request to repaint a portion of an application's minimized (iconic) window.
WM_PARENTNOTIFY	Sent to the parent of a child window when the child window is created or destroyed.
WM_QUERYDRAGICON	Sent when the user is about to drag a minimized (iconic) window.
WM_QUERYENDSESSION	Sent when the user chooses the End Session command.
WM_QUERYNEWPALETTE	Sent when a window is about to receive the input focus to allow it to realize its logical color palette.

Message	Description
WM_QUERYOPEN	Sent to an icon when the user requests that the icon be opened into a window.
WM_QUIT	Indicates a request to terminate an application.
WM_SETFOCUS	Sent after a window receives the input focus.
WM_SETFONT	Changes the font used by a control for drawing text.
WM_SETREDRAW	Sets or clears the redraw flag, which determines whether or not updates to a control are displayed.
WM_SETTEXT	Sets the text of a window.
WM_SHOWWINDOW	Sent whenever a window is to be hidden or shown.
WM_SIZE	Sent after the size of a window has been changed.

5.2 Initialization Messages

Initialization messages are sent by Windows when an application creates a menu or dialog box. The following list briefly describes each initialization message:

Message	Description
WM_INITDIALOG	Sent immediately before a dialog box is displayed.
WM_INITMENU	Requests that a menu be initialized.
WM_INITMENUPOPUP	Sent immediately before a pop-up menu is displayed.

5.3 Input Messages

Input messages are sent by Windows when an application receives input through the mouse, keyboard, scroll bars, or system timer. The following list briefly describes each input message:

Message	Description
WM_CHAR	Results when a WM_KEYUP and a WM_KEYDOWN message are translated.
WM_CHARTOITEM	Sent by a list box with the LBS_WANTKEYBOARDINPUT style to its owner in response to a WM_CHAR message.
WM_COMMAND	Sent when the user selects an item from a menu, when a control passes a message to its parent window, or when an accelerator key stroke is translated.
WM_DEADCHAR	Results when a WM_KEYUP and a WM_KEYDOWN message are translated.
WM_HSCROLL	Sent when the user clicks the horizontal scroll bar with the mouse.
WM_KEYDOWN	Sent when a nonsystem key is pressed.
WM_KEYUP	Sent when a nonsystem key is released.
WM_LBUTTONDBLCLK	Sent when the user double-clicks the left mouse button.
WM_LBUTTONDOWN	Sent when the user presses the left mouse button.
WM_LBUTTONUP	Sent when the user releases the left mouse button.
WM_MBUTTONDBLCLK	Sent when the user double-clicks the middle mouse button.
WM_MBUTTONDOWN	Sent when the user presses the middle mouse button.
WM_MBUTTONUP	Sent when the user releases the middle mouse button.
WM_MOUSEACTIVATE	Sent when the cursor is in an inactive window and any mouse button is pressed.
WM_MOUSEMOVE	Sent when the user moves the mouse.

Message	Description
WM_RBUTTONDBLCLK	Sent when the user double-clicks the right mouse button.
WM_RBUTTONDOWN	Sent when the user presses the right mouse button.
WM_RBUTTONUP	Sent when the user releases the right mouse button.
WM_SETCURSOR	Sent when mouse input is not captured and the mouse causes cursor movement within a window.
WM_TIMER	Sent when the time limit set for a given timer has elapsed.
WM_VKEYTOITEM	Sent by a list box with the LBS_WANTKEYBOARDINPUT style to its owner in response to a WM_CHAR message.
WM_VSCROLL	Sent when the user clicks the vertical scroll bar with the mouse.

5.4 System Messages

System messages are sent by Windows to an application when a user accesses a window's System menu, scroll bars, or size box. Although an application can process these messages, most applications pass them on to the **DefWindowProc** function for default processing. The following list briefly describes each system message:

Message	Description
WM_SYSCHAR	Results when a WM_SYSKEYUP and a WM_SYSKEYDOWN message are translated.
WM_SYSCOMMAND	Sent when the user selects a command from the System menu.
WM_SYSDEADCHAR	Results when a WM_SYSKEYUP and a WM_SYSKEYDOWN message are translated.
WM_SYSKEYDOWN	Sent when the user holds down the ALT key and then presses another key.

Message	Description
WM_SYSKEYUP	Sent when the user releases a key that was pressed while the ALT key was held down.

5.5 Clipboard Messages

Clipboard messages are sent by Windows to an application when other applications try to access a window's clipboard. The following list briefly describes each clipboard message:

Message	Description
WM_ASKCBFORMATNAME	Requests the name of the CF_OWNERDISPLAY format.
WM_CHANGECBCHAIN	Notifies viewing-chain members of a change in the chain.
WM_DESTROYCLIPBOARD	Signals that the contents of the clipboard are being destroyed.
WM_DRAWCLIPBOARD	Signals an application to notify the next application in the chain of a clipboard change.
WM_HSCROLLCLIPBOARD	Requests horizontal scrolling for the CF_OWNERDISPLAY format.
WM_PAINTCLIPBOARD	Requests painting of the CF_OWNERDISPLAY format.
WM_RENDERALLFORMATS	Notifies the clipboard owner that it must render the clipboard data in all possible formats.
WM_RENDERFORMAT	Notifies the clipboard owner that it must format the last data copied to the clipboard.
WM_SIZECLIPBOARD	Notifies the clipboard owner that the clipboard application's window size has changed.
WM_VSCROLLCLIPBOARD	Requests vertical scrolling for the CF_OWNERDISPLAY format.

5.6 System-Information Messages

System-information messages are sent by Windows when an application or a user makes a system-wide change that affects other applications. The following list briefly describes each system-information message:

Message	Description
WM_COMPACTING	Sent to all top-level windows when Windows requires too much system time compacting memory, indicating that system memory is low.
WM_DEVMODECHANGE	Sent to all top-level windows when the user changes device-mode settings.
WM_FONTCHANGE	Sent when the pool of font resources changes.
WM_PALETTECHANGED	Notifies all windows that the system color palette has changed.
WM_SPOOLERSTATUS	Sent from Print Manager whenever a job is added to or removed from the Print Manager queue.
WM_SYSCOLORCHANGE	Sent to all top-level windows when a change is made in the system color setting.
WM_TIMECHANGE	Sent when an application makes a change or set of changes to the system time.
WM_WININICHANGE	Sent when the Windows initialization file, WIN.INI, changes.

5.7 Control Messages

Control messages are predefined window messages that direct a control to carry out a specified task. Applications send control messages to a control by using the **SendMessage** function. The control carries out the specified task and returns a value that indicates the result.

The following messages apply to all controls:

Message	Description
WM_NEXTDLGCTL	Sent to a dialog box's window function, to alter the control focus.
WM_GETFONT	Retrieves the current font used by a control for drawing text.
WM_SETFONT	Changes the font used by a control for drawing text.

Sections 5.7.1 through 5.7.5 briefly describe the control messages for each control class.

5.7.1 Button-Control Messages

Button-control messages are sent by an application to a button control. The following list briefly describes each button-control message:

Message	Description
BM_GETCHECK	Determines whether a radio button or check box is checked.
BM_GETSTATE	Returns nonzero if the cursor is over the button and the user presses the mouse button or the SPACEBAR.
BM_SETCHECK	Checks or removes the checkmark from a radio button or check box.
BM_SETSTATE	Highlights a button or check box.
BM_SETSTYLE	Alters the style of a button.
DM_GETDEFID	Retrieves the ID of the default push-button control for a dialog box.
DM_SETDEFID	Changes the default push-button control ID for a dialog box.

5.7.2 Edit-Control Messages

Edit-control messages are sent by an application to an edit control. In addition to the messages described below, the WM_ENABLE, WM_GETTEXT, WM_GETTEXTLENGTH, WM_KILLFOCUS, WM_SETFOCUS, WM_SETREDRAW, and WM_SETTEXT window messages can be used. The following list briefly describes each edit-control message:

Message	Description
EM_CANUNDO	Determines whether or not an edit control can respond correctly to an EM_UNDO message.
EM_EMPTYUNDOBUFFER	Disables an edit control's ability to undo the last edit.
EM_FMTLINES	Directs the edit control to add or remove the end-of-line character from wordwrapped text lines.
EM_GETHANDLE	Returns the data handle of the buffer used to hold the contents of the control window.
EM_GETLINE	Copies a line from the edit control.
EM_GETLINECOUNT	Returns the number of lines of text in the edit control.
EM_GETMODIFY	Returns the current value of the modify flag for a given edit control. The flag is set by the control if the user enters or modifies text within the control.
EM_GETRECT	Returns the formatting rectangle of the edit control.
EM_GETSEL	Returns the starting and ending character positions of the current selection.
EM_LIMITTEXT	Limits the length of the text (in bytes) the user may enter.
EM_LINEFROMCHAR	Returns the line number of the line that contains the character whose position (indexed from the beginning of the text) is specified by the *wParam* parameter.
EM_LINEINDEX	Returns the number of character positions that occur before the first character in a given line.
EM_LINELENGTH	Returns the length of a line (in bytes) in the edit control's text buffer.
EM_LINESCROLL	Scrolls the contents of the edit control by the given number of lines.

Message	Description
EM_REPLACESEL	Replaces the current selection with new text.
EM_SETHANDLE	Establishes the text buffer used to hold the contents of the edit-control window.
EM_SETMODIFY	Sets the modify flag for a given edit control.
EM_SETPASSWORDCHAR	Changes the password character for an edit control created with the ES_PASSWORD styles.
EM_SETRECT	Sets the formatting rectangle for an edit control.
EM_SETRECTNP	Identical to EM_SETRECT, except that the control is *not* repainted.
EM_SETSEL	Selects all characters in the current text that are within the starting and ending character positions given by the *lParam* parameter.
EM_SETTABSTOPS	Sets tab-stop positions in a multiline edit control.
EM_SETWORDBREAK	Informs a multiline edit control that Windows has replaced the default word-break function with an application-supplied word-break function.
EM_UNDO	Undoes the last edit in an edit control.
WM_CLEAR	Deletes the current selection.
WM_COPY	Sends the current selection to the clipboard in CF_TEXT format.
WM_CUT	Sends the current selection to the clipboard in CF_TEXT format, and then deletes the selection from the control window.
WM_PASTE	Inserts the data from the clipboard into the control window at the current cursor position.
WM_UNDO	Undoes the previous action.

5.7.3 List-Box Messages

List-box messages are sent by an application to a list box. The following list briefly describes each list-box message:

Message	Description
LB_ADDSTRING	Adds a string to the list box.
LB_DELETESTRING	Deletes a string from the list box.
LB_DIR	Adds a list of the files from the current directory to the list box.
LB_FINDSTRING	Finds the first string in the list box which matches prefix text.
LB_GETCOUNT	Returns a count of the number of items in the list box.
LB_GETCURSEL	Returns the index of the currently selected item, if any.
LB_GETHORIZONTALEXTENT	Retrieves the width by which a list box can be scrolled horizontally.
LB_GETITEMDATA	Retrieves a 32-bit value associated with an item in an owner-draw list box.
LB_GETITEMRECT	Retrieves the coordinates of the rectangle that bounds a list-box item.
LB_GETSEL	Returns the selection state of an item.
LB_GETSELCOUNT	Returns the total number of selected items in a multiselection list box.
LB_GETSELITEMS	Retrieves the indexes of the selected items in a multiselection list box.
LB_GETTEXT	Copies a string from the list box into a buffer.
LB_GETTEXTLEN	Returns the length of a string in the list box.
LB_GETTOPINDEX	Returns the index of the first visible item in a list box.
LB_INSERTSTRING	Inserts a string in the list box.

Message	Description
LB_RESETCONTENT	Removes all strings from a list box and frees any memory allocated for those strings.
LB_SELECTSTRING	Changes the current selection to the first string that has the specified prefix.
LB_SELITEMRANGE	Selects one or more consecutive items in a multiple-selection list box.
LB_SETCOLUMNWIDTH	Sets the width in pixels of all columns in a multicolumn list box.
LB_SETCURSEL	Selects a string and scrolls it into view, if necessary.
LB_SETHORIZONTALEXTENT	Sets the width by which a list box can be scrolled horizontally.
LB_SETITEMDATA	Sets a 32-bit value associated with an item in an owner-draw list box.
LB_SETSEL	Sets the selection state of a string.
LB_SETTABSTOPS	Sets tab-stop positions in a list box.
LB_SETTOPINDEX	Sets the first visible item in a list box to the item identified by an index.

5.7.4 Combo-Box Messages

Combo-box messages are sent by an application to a combo box. The following list briefly describes each combo-box message:

Message	Description
CB_ADDSTRING	Adds a string to the list box of a combo box.
CB_DELETESTRING	Deletes a string from the list box of a combo box.
CB_DIR	Adds a list of the files from the current directory to the combo box.
CB_FINDSTRING	Finds the first string in the combo-box list box which matches a prefix.

Message	Description
CB_GETCOUNT	Returns a count of the number of items in the combo box.
CB_GETCURSEL	Returns the index of the currently selected item, if any.
CB_GETEDITSEL	Returns the starting and ending positions of the selected text in the edit control of a combo box.
CB_GETITEMDATA	Retrieves a 32-bit value associated with an item in an owner-draw combo box.
CB_GETLBTEXT	Copies a string from the list box of a combo box into a buffer.
CB_GETLBTEXTLEN	Returns the length of a string in the list box of a combo box.
CB_INSERTSTRING	Inserts a string in the combo box.
CB_LIMITTEXT	Limits the length of the text that the user may enter into the edit control of a combo box.
CB_RESETCONTENT	Removes all strings from a combo box and frees any memory allocated for those strings.
CB_SELECTSTRING	Changes the current selection to the first string that has the specified prefix. The text in the edit control is changed to reflect the new selection.
CB_SETCURSEL	Selects a string and scrolls it into view, if necessary.
CB_SETEDITSEL	Selects all characters in the edit control that are within specified starting and ending positions.
CB_SETITEMDATA	Sets a 32-bit value associated with an item in an owner-draw combo box.
CB_SHOWDROPDOWN	Shows or hides a drop-down list box in a combo box.

5.7.5 Owner Draw–Control Messages

Owner draw–control messages notify the owner of a control created with the OWNERDRAW style that the control needs to be drawn and to provide information about the drawing required. The following list briefly describes these messages:

Message	Description
WM_COMPAREITEM	Determines which of two items sorts above the other in a sorted owner-draw list box or combo box.
WM_DELETEITEM	Indicates that an item in an owner-draw list box or combo box has been deleted.
WM_DRAWITEM	Indicates that an owner-draw control needs to be redrawn.
WM_MEASUREITEM	Requests the dimensions of an owner-draw combo box, list box, or menu item.

5.8 Notification Messages

Notification messages notify a control's parent window of actions that occur within a control. Sections 5.8.1 through 5.8.4 briefly describe the notification messages for each notification class.

Controls use the WM_COMMAND message to notify the parent window of actions that occur within the control. The *wParam* parameter of the WM_COMMAND message contains the control ID; the low-order word of the *lParam* parameter contains the control-window handle; and the high-order word of *lParam* contains the control notification code.

5.8.1 Button Notification Codes

The following notification codes apply to buttons:

Message	Description
BN_CLICKED	Indicates that the button has been clicked.
BN_DOUBLECLICKED	Indicates that the user has double-clicked an owner-draw or radio button.

5.8.2 Edit-Control Notification Codes

The following notification codes apply to edit controls:

Message	Description
EN_CHANGE	Indicates that the user has taken some action that may have changed the content of the text.
EN_ERRSPACE	Indicates that the edit control is out of space.
EN_HSCROLL	Indicates that the user has clicked the edit control's horizontal scroll bar with the mouse; the parent window is notified before the screen is updated.
EN_KILLFOCUS	Indicates that the edit control has lost the input focus.
EN_MAXTEXT	Specifies that the current insertion has exceeded a specified number of characters for the edit control.
EN_SETFOCUS	Indicates that the edit control has obtained the input focus.
EN_UPDATE	Specifies that the edit control will display altered text.
EN_VSCROLL	Indicates that the user has clicked the edit control's vertical scroll bar with the mouse; the parent window is notified before the screen is updated.

5.8.3 List-Box Notification Codes

The following notification codes apply only to list-box controls that have LBS_NOTIFY style:

Message	Description
LBN_DBLCLK	Sent when the user double-clicks a string with the mouse.
LBN_ERRSPACE	Sent when the system is out of memory.
LBN_KILLFOCUS	Indicates that a list box has lost input focus.
LBN_SELCHANGE	Sent when the selection has been changed.
LBN_SETFOCUS	Indicates that the list box has received input focus.

5.8.4 Combo-Box Notification Codes

The following notification codes apply to combo boxes:

Message	Description
CBN_DBLCLK	Sent when the user double-clicks a string with the mouse.
CBN_DROPDOWN	Informs the owner of the combo box that its list box is about to be dropped down.
CBN_EDITCHANGE	Indicates that the user has altered text in the edit control.
CBN_EDITUPDATE	Indicates that the edit control will display altered text.
CBN_ERRSPACE	Sent when the system is out of memory.
CBN_KILLFOCUS	Indicates that a combo box has lost input focus.
CBN_SELCHANGE	Sent when the selection has been changed.
CBN_SETFOCUS	Indicates that the combo box has received input focus.

5.9 Scroll-Bar Messages

There are two messages in the scroll-bar group: WM_HSCROLL and WM_VSCROLL. Scroll-bar controls send these messages to their parent windows whenever the user clicks in the control. The *wParam* parameter contains the same values as those defined for the scrolling messages of a standard window. The high-order word of the *lParam* parameter contains the window handle of the scroll-bar control.

5.10 Nonclient-Area Messages

Nonclient-area messages are sent by Windows to create and maintain the nonclient area of an application's window. Normally, applications do not process these messages, but send them on to the **DefWindowProc** function for processing. The following list briefly describes each nonclient-area message:

Message	Description
WM_NCACTIVATE	Sent to a window when its caption bar or icon needs to be changed to indicate an active or inactive state.
WM_NCCALCSIZE	Sent when the size of a window's client area needs to be calculated.
WM_NCCREATE	Sent prior to the WM_CREATE message when a window is first created.
WM_NCDESTROY	Sent after the WM_DESTROY message.
WM_NCHITTEST	Sent to the window that contains the cursor (unless a window has captured the mouse).
WM_NCLBUTTONDBLCLK	Sent to a window when the left mouse button is double-clicked while the cursor is in a nonclient area of the window.
WM_NCLBUTTONDOWN	Sent to a window when the left mouse button is pressed while the cursor is in a nonclient area of the window.
WM_NCLBUTTONUP	Sent to a window when the left mouse button is released while the cursor is in a nonclient area of the window.
WM_NCMBUTTONDBLCLK	Sent to a window when the middle mouse button is double-clicked while the cursor is in a nonclient area of the window.
WM_NCMBUTTONDOWN	Sent to a window when the middle mouse button is pressed while the cursor is in a nonclient area of the window.
WM_NCMBUTTONUP	Sent to a window when the left mouse button is released while the cursor is in a nonclient area of the window.

Message	Description
WM_NCMOUSEMOVE	Sent to a window when the cursor is moved in a nonclient area of the window.
WM_NCPAINT	Sent to a window when its border needs painting.
WM_NCRBUTTONDBLCLK	Sent to a window when the right mouse button is double-clicked while the cursor is in a nonclient area of the window.
WM_NCRBUTTONDOWN	Sent to a window when the right mouse button is pressed while the cursor is in a nonclient area of the window.
WM_NCRBUTTONUP	Sent to a window when the right mouse button is released while the cursor is in a nonclient area of the window.

5.11 Multiple Document Interface Messages

Windows multiple document interface (MDI) provides applications with a standard interface for displaying multiple documents within the same instance of an application. An MDI application creates a frame window which contains a client window in place of its client area. The application creates an MDI client window by calling **CreateWindow** with the **MDICLIENT** class and passing a **CLIENTCREATESTRUCT** data structure as the function's *lpParam* parameter. This client window in turn can own multiple child windows, each of which displays a separate document. An MDI application controls these child windows by sending messages to its client window. The following briefly describes these MDI messages:

Message	Description
WM_MDIACTIVATE	Activates a child window.
WM_MDICASCADE	Arranges child windows in a cascade format.
WM_MDICREATE	Creates a child window.
WM_MDIDESTROY	Closes a child window.
WM_MDIGETACTIVE	Returns the current active MDI child window.

Message	Description
WM_MDIICONARRANGE	Arranges all minimized child windows.
WM_MDIMAXIMIZE	Maximizes an MDI child window.
WM_MDINEXT	Activates the next child window.
WM_MDIRESTORE	Restores a child window from a maximized or minimized state.
WM_MDISETMENU	Replaces the menu of an MDI frame window, the Window pop-up menu, or both.
WM_MDITILE	Arranges all child windows in a tiled format.

5.12 Summary

Windows messages provide the means of communication between the Windows system and applications, as well as among applications, in a nonpreemptive multitasking environment. For more information on topics related to Windows messages, see the following:

Topic	Reference
Message-processing functions	*Reference, Volume 1*: Chapter 1, "Window Manager Interface Functions"
Function descriptions	*Reference, Volume 1*: Chapter 4, "Functions Directory"
Message descriptions	*Reference, Volume 1*: Chapter 6, "Messages Directory"
Windows data types and structures	*Reference, Volume 2*: Chapter 7, "Data Types and Structures"
Dynamic data exchange	*Reference, Volume 2*: Chapter 15, "Windows DDE Protocol Definition"
	Guide to Programming: Chapter 22, "Dynamic Data Exchange"
General information on Windows programming	*Guide to Programming*: Chapter 1, "An Overview of the Windows Environment"

Chapter 6

Messages Directory

Microsoft Windows communicates with applications through formatted window messages. These messages are sent to an application's window function for processing.

Some messages return values that contain information about the success of the message or other data needed by an application. To obtain the return value, the application must call **SendMessage** to send the message to a window. This function does not return until the message has been processed. If the application does not require the return value of the message, it may call **PostMessage** to send the message. This function places a message in a window's application queue and then returns immediately. If a message does not have a return value, then the application may use either function to send the message, unless indicated otherwise in the message description.

A message consists of three parts: a message number, a word parameter, and a long parameter. Message numbers are identified by predefined message names. The message names begin with letters that suggest the meaning or origin of the message. The word and long parameters, named *wParam* and *lParam* respectively, contain values that depend on the message number.

The *lParam* parameter often contains more than one type of information. For example, the high-order word may contain a handle to a window and the low-order word may contain an integer value. The **HIWORD** and **LOWORD** utility macros can be used to extract the high- and low-order words of the *lParam* parameter. The **HIBYTE** and **LOBYTE** utility macros can also be used with **HIWORD** and **LOWORD** to access any of the bytes. Casting can also be used.

There are four ranges of message numbers, as shown in the following list:

Range	Meaning
0 to WM_USER – 1	Reserved for use by Windows.
WM_USER to 0x7FFF	Integer messages for use by applications.
0x8000 to 0xBFFF	Reserved for use by Windows.
0xC000 to 0xFFFF	String messages for use by applications.

Message numbers in the first range (0 to WM_USER – 1) are defined by Windows. Values in this range that are not explicitly defined are reserved for future use by Windows. This chapter describes messages in this range.

Message numbers in the second range (WM_USER to 7FFF) can be defined and used by an application to send messages within the application. These messages should *not* be sent to other applications unless the applications have been designed to exchange messages and to attach the same meaning to the message numbers.

Message numbers in the third range (8000 to BFFF) are reserved for future use by Windows.

Message numbers in the fourth range (C000 to FFFF) are defined at run time when an application calls the **RegisterWindowMessage** function to obtain a message number for a string. All applications that register the identical string can use the associated message number for exchanging messages with each other. The actual message number, however, is not a constant and cannot be assumed to be the same in different window sessions.

This chapter lists messages in alphabetical order. For more information about messages, see Chapter 5, "Messages Overview."

BM_GETCHECK

This message determines whether a radio button or check box is checked.

Parameter	Description
wParam	Is not used.
lParam	Is not used.

Return Value The return value is nonzero if the radio button or check box is checked. Otherwise, it is zero. The BM_GETCHECK message always returns zero for a push button.

BM_GETSTATE

This message determines the state of a button control when the user presses a mouse button or the SPACEBAR.

Parameter	Description
wParam	Is not used.
lParam	Is not used.

Return Value The BM_GETSTATE message returns a nonzero value if one of the following occurs:

- A push button is highlighted.
- The user presses a mouse button or the SPACEBAR when a button has the input focus.
- The user presses a mouse button when the cursor is over a button.

Otherwise, BM_GETSTATE returns zero.

BM_SETCHECK

This message checks or removes the checkmark from a radio button or check box.

Parameter	Description
wParam	Specifies whether to place or remove a checkmark inside the button or box. If the *wParam* parameter is nonzero, a checkmark is placed; if it is zero, the checkmark (if any) is removed. For three-state buttons, if *wParam* is 1, a checkmark is placed beside the button. If *wParam* is 2, the button is grayed. If *wParam* is zero, the button is returned to its normal state (no checkmark or graying).
lParam	Is not used.

Comments The BM_SETCHECK message has no effect on push buttons.

BM_SETSTATE

This message displays a button or check box.

Parameter	Description
wParam	Specifies the highlighting action to be taken. If the *wParam* parameter is nonzero, the button is highlighted (the interior is drawn using inverse video). If *wParam* is zero, the button is drawn in its regular state.
lParam	Is not used.

Comments Push buttons cannot be highlighted.

BM_SETSTYLE

This message alters the style of buttons. If the style contained in the *wParam* parameter differs from the existing style, the button is redrawn in the new style.

Parameter	Description
wParam	Specifies the style value. For a complete description of possible button styles, see Table 6.1, "Button Styles."
lParam	Specifies whether or not the buttons are to be redrawn. If *lParam* is zero, the buttons will not be redrawn. If *lParam* is nonzero, they will be redrawn.

Comments Table 6.1 describes the available button styles:

Table 6.1 Button Styles

Value	Meaning
BS_AUTOCHECKBOX	Identical to BS_CHECKBOX, except that the button automatically toggles its state whenever the user clicks it.
BS_AUTORADIOBUTTON	Identical to BS_RADIOBUTTON, except that the button is checked, the application is notified by BN_CLICKED, and the checkmarks are removed from all other radio buttons in the group.
BS_AUTO3STATE	Identical to BS_3STATE, except that the button automatically toggles its state when the user clicks it.
BS_CHECKBOX	Designates a box that may be checked; its border is bold when the user clicks the button. Any text appears to the right of the box.
BS_DEFPUSHBUTTON	Designates a button with a bold border. This button represents the default user response. Any text is displayed within the button. Windows sends a message to the parent window when the user clicks the button.
BS_GROUPBOX	Designates a rectangle into which other buttons are grouped. Any text is displayed in the rectangle's upper-left corner.
BS_LEFTTEXT	Causes text to appear on the left side of the radio button or check-box button. Use this style with the BS_CHECKBOX, BS_RADIOBUTTON, or BS_3STATE styles.
BS_OWNERDRAW	Designates an owner-draw button. The parent window is notified when the button is clicked. Notification includes a request to paint, invert, and disable the button.
BS_PUSHBUTTON	Designates a button that contains the given text. The control sends a message to its parent window whenever the user clicks the button.
BS_RADIOBUTTON	Designates a small circular button that can be checked; its border is bold when the user clicks the button. Any text appears to the right of the button. Typically, two or more radio buttons are grouped together to represent mutually exclusive choices, so no more than one button in the group is checked at any time.
BS_3STATE	Identical to BS_CHECKBOX, except that the box can be grayed as well as checked. The grayed state typically is used to show that a check box has been disabled.

BN_CLICKED

This code specifies that the user has clicked a button. The parent window receives the code through a WM_COMMAND message from a button control.

Parameter	Description
wParam	Specifies the control ID.
lParam	Contains a handle that identifies the button control in its low-order word and the BN_CLICKED notification code in its high-order word.

Comments Disabled buttons will not send a BN_CLICKED notification message to a parent window.

BN_DOUBLECLICKED

This code specifies that the user has double-clicked a button. The control's parent window receives this code through a WM_COMMAND message from a button control.

Parameter	Description
wParam	Specifies the control ID.
lParam	Contains a handle that identifies the button control in its low-order word and the BN_DOUBLECLICKED notification code in its high-order word.

Comments This code applies to buttons with the BS_RADIOBUTTON and BS_OWNERDRAW styles only.

CB_ADDSTRING 3.0

This message adds a string to the list box of a combo box. If the list box is not sorted, the string is added to the end of the list. If the list box is sorted, the string is inserted into the list after sorting.

This message removes any existing list-box selections.

Parameter	Description
wParam	Is not used.
lParam	Points to the null-terminated string that is to be added. If the combo box was created with an owner-draw style but without the CBS_HASSTRINGS style, the *lParam* parameter is an application-supplied 32-bit value that is stored by the combo box instead of the pointer to the string.

Return Value The return value is the index to the string in the list box. The return value is CB_ERR if an error occurs; the return value is CB_ERRSPACE if insufficient space is available to store the new string.

Comments If an owner-draw combo box was created with the CBS_SORT style but not the CBS_HASSTRINGS style, the WM_COMPAREITEM message is sent one or more times to the owner of the combo box so that the new item can be properly placed in the list box.

CB_DELETESTRING 3.0

This message deletes a string from the list box.

Parameter	Description
wParam	Contains an index to the string that is to be deleted.
lParam	Is not used.

Return Value The return value is a count of the strings remaining in the list. The return value is CB_ERR if *wParam* does not specify a valid index.

Comments If the combo box was created with an owner-draw style but without the CBS_HASSTRINGS style, a WM_DELETEITEM message is sent to the owner of the combo box so the application can free additional data associated with the item (through the *lParam* parameter of the CB_ADDSTRING or CB_INSERTSTRING message).

CB_DIR 3.0

This message adds a list of the files from the current directory to the list box. Only files with the attributes specified by the *wParam* parameter and that match the file specification given by the *lParam* parameter are added.

Parameter	Description
wParam	Contains a DOS attribute value. For a list of the DOS attributes, see the **DlgDirList** function in Chapter 4, "Functions Directory."
lParam	Points to a file-specification string. The string can contain wildcard characters (for example, *.*).

Return Value The return value is a count of items displayed. The return value is CB_ERR if an error occurs; the return value is CB_ERRSPACE if insufficient space is available to store the new strings.

Comments The return value of the CB_DIR message is one less than the return value of the CB_GETCOUNT message.

CB_FINDSTRING 3.0

This message finds the first string in the list box of a combo box which matches the given prefix text.

Parameter	Description
wParam	Contains the index of the item before the first item to be searched. When the search reaches the bottom of the list box it continues from the top of the list box back to the item specified by *wParam*. If the *wParam* parameter is –1, the entire list box is searched from the beginning.
lParam	Points to the prefix string. The string must be null-terminated.

Return Value The return value is the index of the matching item or CB_ERR if the search was unsuccessful.

Comments If the combo box was created with an owner-draw style but without the CBS_HASSTRINGS style, this message returns the index of the item whose long value (supplied as the *lParam* parameter of the CB_ADDSTRING or CB_INSERTSTRING message) matches the value supplied as the *lParam* parameter of CB_FINDSTRING.

CB_GETCOUNT 3.0

This message returns a count of the items in a list box of a combo box.

Parameter	Description
wParam	Is not used.
lParam	Is not used.

Return Value The return value is a count of the items in the list box of a combo box.

CB_GETCURSEL 3.0

This message returns the index of the currently selected item, if any, in the list box of a combo box.

Parameter	Description
wParam	Is not used.
lParam	Is not used.

Return Value The return value is the index of the currently selected item. It is CB_ERR if no item is selected.

CB_GETEDITSEL 3.0

This message returns the starting and ending positions of the selected text in the edit control of a combo box.

Parameter	Description
wParam	Is not used.
lParam	Is not used.

Return Value The return value is a long integer containing the starting position in the low-order word and the ending position in the high-order word. If this message is sent to a combo box without an edit control, the return value is CB_ERR.

CB_GETITEMDATA 3.0

This message retrieves the application-supplied 32-bit value associated with the specified combo-box item. If the item is in an owner-draw combo box created without the CBS_HASSTRINGS style, this 32-bit value was contained in the *lParam* parameter of the CB_ADDSTRING or CB_INSERTSTRING message that added the item to the combo box. Otherwise, it was the value in the *lParam* parameter of a CB_SETITEMDATA message.

Parameter	Description
wParam	Contains an index to the item.
lParam	Is not used.

Return Value The return value is the 32-bit value associated with the item, or CB_ERR if an error occurs.

CB_GETLBTEXT 3.0

This message copies a string from the list box of a combo box into a buffer.

Parameter	Description
wParam	Contains the index of the string to be copied.
lParam	Points to a buffer that is to receive the string. The buffer must have sufficient space for the string and a terminating null character.

Return Value The return value is the length of the string in bytes, excluding the terminating null character. If *wParam* does not specify a valid index, the return value is CB_ERR.

Comments If the combo box was created with an owner-draw style but without the CBS_HASSTRINGS style, the buffer pointed to by the *lParam* parameter of the message receives the 32-bit value associated with the item through the *lParam* parameter of the CB_ADDSTRING or CB_INSERTSTRING message.

CB_GETLBTEXTLEN 3.0

This message returns the length of a string in the list box of a combo box.

Parameter	Description
wParam	Contains the index of the string.
lParam	Is not used.

Return Value The return value is the length of the string in bytes, excluding the terminating null character. If *wParam* does not specify a valid index, the return value is CB_ERR.

CB_INSERTSTRING 3.0

This message inserts a string into the list box of a combo box. No sorting is performed.

Parameter	Description
wParam	Contains an index to the position that will receive the string. If the *wParam* parameter is –1, the string is added to the end of the list.
lParam	Points to the null-terminated string that is to be inserted. If the combo box was created with an owner-draw style but without the CBS_HASSTRINGS style, the *lParam* parameter is an application-supplied 32-bit value that is stored by the combo box instead of the pointer to the string.

Return Value The return value is the index of the position at which the string was inserted. The return value is CB_ERR if an error occurs; the return value is CB_ERRSPACE if insufficient space is available to store the new string.

CB_LIMITTEXT 3.0

This message limits the length (in bytes) of the text that the user may enter into the edit control of a combo box.

Parameter	Description
wParam	Specifies the maximum number of bytes which the user can enter.
lParam	Is not used.

Return Value The return value is TRUE if the message is successful; otherwise, it is FALSE. If this message is sent to a combo box without an edit control, the return value is CB_ERR.

CB_RESETCONTENT 3.0

This message removes all strings from the list box of a combo box and frees any memory allocated for those strings.

Parameter	Description
wParam	Is not used.
lParam	Is not used.

Comments If the combo box was created with an owner-draw style but without the CBS_HAS-STRINGS style, the owner of the combo box receives a WM_DELETEITEM message for each item in the combo box.

CB_SELECTSTRING 3.0

This message selects the first string in the list box of a combo box that matches the specified prefix. The text in the edit control of the combo box is changed to reflect the new selection.

Parameter	Description
wParam	Contains the index of the item before the first item to be searched. When the search reaches the bottom of the list box it continues from the top of the list box back to the item specified by *wParam*. If the *wParam* parameter is –1, the entire list box is searched from the beginning.
lParam	Points to the prefix string. The string must have a null-terminating character.

Return Value The return value is the index of the newly selected item. If the search was unsuccessful, the return value is CB_ERR and the current selection is not changed.

Comments A string is selected only if its initial characters (from the starting point) match the characters in the prefix string.

If the combo box was created with an owner-draw style but without the CBS_HAS-STRINGS style, this message returns the index of the item whose long value (supplied as the *lParam* parameter of the CB_ADDSTRING or CB_INSERTSTRING message) matches the value supplied as the *lParam* parameter of CB_FINDSTRING.

CB_SETCURSEL 3.0

This message selects a string in the list box of a combo box and scrolls it into view if the list box is visible, and the text in the combo-box edit control or static-text control is changed to reflect the new selection. When the new string is selected, the list box removes the highlight from the previously selected string.

Parameter	Description
wParam	Contains the index of the string that is to be selected. If *wParam* is –1, the list box is set to have no selection.
lParam	Is not used.

Return Value If the index specified by *wParam* is not valid, the return value is CB_ERR and the current selection is not changed.

CB_SETEDITSEL 3.0

This message selects all characters in the edit control of a combo box that are within the starting and ending character positions specified by the *lParam* parameter.

Parameter	Description
wParam	Is not used.
lParam	Specifies the starting position in the low-order word and the ending position in the high-order word.

Return Value The return value is TRUE if the message is successful; otherwise, it is FALSE. If this message is sent to a combo box without an edit control, the return value is CB_ERR.

CB_SETITEMDATA 3.0

This message sets the 32-bit value associated with the specified item in a combo box. If the item is in an owner-draw combo box created without the CBS_HASSTRINGS style, this message replaces the 32-bit value that was contained in the *lParam* parameter of the CB_ADDSTRING or CB_INSERTSTRING message that added the item to the combo box.

Parameter	Description
wParam	Contains an index to the item.
lParam	Contains the new value to be associated with the item.

Return Value The return value is CB_ERR if an error occurs.

CB_SHOWDROPDOWN 3.0

This message shows or hides the drop-down list box on a combo box created with the CBS_DROPDOWN or CBS_DROPDOWNLIST style.

Parameter	Description
wParam	If TRUE, displays the list box if it is not already visible. If FALSE, hides the list box if it is visible.
lParam	Not used.

CBN_DBLCLK 3.0

This code specifies that the user has double-clicked a string in the list box of a combo box. The control's parent window receives this code through a WM_COMMAND message from the control.

Parameter	Description
wParam	Specifies the control ID of the combo box.
lParam	Contains the combo-box window handle in its low-order word and the CBN_DBLCLK code in its high-order word.

Comments This message can only occur for a combo box with a list box that is always visible. For combo boxes with drop-down list boxes, a single closes the list box and so a double-click cannot occur.

CBN_DROPDOWN 3.0

This code specifies that the list box of a combo box will be dropped down. It is sent just before the combo-box list box is made visible. The control's parent window receives this code through a WM_COMMAND message from the control.

Parameter	Description
wParam	Specifies the control ID of the combo box.
lParam	Contains the combo-box window handle in its low-order word and the CBN_DROPDOWN code in the high-order word.

Comments This message does not occur if the combo box does not contain a drop-down list box.

CBN_EDITCHANGE 3.0

This code indicates that the user has taken an action that may have altered the text in the edit control of a combo box. It is sent after Windows updates the display (unlike the CBN_EDITUPDATE code). The control's parent window receives this code through a WM_COMMAND message from the control.

Parameter	Description
wParam	Specifies the control ID of the combo box.
lParam	Contains the combo-box window handle in its low-order word and the CBN_EDITCHANGE code in its high-order word.

Comments This message does not occur if the combo box does not contain an edit control.

CBN_EDITUPDATE 3.0

This code specifies that a combo box containing an edit control will display altered text. The control's parent window receives this code through a WM_COMMAND message from the control.

Parameter	Description
wParam	Specifies the control ID of the combo box.
lParam	Contains the combo-box window handle in its low-order word and the CBN_EDITUPDATE code in its high-order word.

Comments This message does not occur if the combo box does not contain an edit control.

CBN_ERRSPACE 3.0

This code specifies that the combo-box list-box control cannot allocate enough memory to meet a specific request. The control's parent window receives this code through a WM_COMMAND message from the control.

Parameter	Description
wParam	Specifies the control ID of the combo box.
lParam	Contains the combo-box window handle in its low-order word and the CBN_ERRSPACE code in its high-order word.

CBN_KILLFOCUS 3.0

This code is sent when a combo box loses input focus. The control's parent window receives this code through a WM_COMMAND message from the control.

Parameter	Description
wParam	Specifies the control ID of the combo box.
lParam	Contains the combo-box window handle in its low-order word and the CBN_KILLFOCUS code in its high-order word.

CBN_SELCHANGE 3.0

This code indicates that the selection in the list box of a combo box has changed either as a result of the user clicking in the list box or entering text in the edit control. The control's parent window receives this code through a WM_COMMAND message from the control.

Parameter	Description
wParam	Specifies the control ID of the combo box.
lParam	Contains the combo-box window handle in its low-order word and the CBN_SELCHANGE code in its high-order word.

CBN_SETFOCUS 3.0

This code is sent when the combo box receives input focus. The control's parent window receives this code through a WM_COMMAND message from the control.

Parameter	Description
wParam	Specifies the control ID of the combo box.
lParam	Contains the combo-box window handle in its low-order word and the CBN_SETFOCUS code in its high-order word.

DM_GETDEFID

This message retrieves the ID of the default push-button control for a dialog box.

Parameter	Description
wParam	Is not used.
lParam	Is not used.

Return Value The return value is a 32-bit value. The high-order word contains DC_HASDEFID if the default button exists; otherwise, it is NULL. The low-order word contains the ID of the default button if the high-order word contains DC_HASDEFID; otherwise, it is zero.

DM_SETDEFID

This message is used by an application to change the default push-button control ID for a dialog box.

Parameter	Description
wParam	Contains the ID of the new default push-button control.
lParam	Is not used.

EM_CANUNDO

This message determines whether an edit control can respond correctly to an EM_UNDO message.

Parameter	Description
wParam	Is not used.
lParam	Is not used.

Return Value The return value is nonzero if the edit control can process the EM_UNDO message correctly. Otherwise, it is zero.

EM_EMPTYUNDOBUFFER 3.0

This message directs an edit control to clear its undo buffer. This disables the edit control's ability to undo the last edit.

Parameter	Description
wParam	Is not used.
lParam	Is not used.

Comments The undo buffer is automatically emptied whenever the edit control receives a WM_SETTEXT or EM_SETHANDLE message.

EM_FMTLINES

This message directs a multiline edit control to add or remove the end-of-line character from word wrapped text lines.

Parameter	Description
wParam	Indicates the disposition of end-of-line characters. If the *wParam* parameter is nonzero, the characters CR CR LF (0D 0D 0A hexadecimal) are placed at the end of wordwrapped lines. If *wParam* is zero, the end-of-line characters are removed from the text.
lParam	Is not used.

Return Value The return value is nonzero if any formatting occurs. Otherwise, it is zero.

Comments Lines that end with a hard return (a carriage return entered by the user) contain the characters CR LF at the end of the line. These lines are not affected by the EM_FMTLINES message.

Notice that the size of the text changes when this message is processed.

EM_GETHANDLE

This message returns the data handle of the buffer that holds the contents of the control window. The handle is always a local handle to a location in the application's data segment.

Parameter	Description
wParam	Is not used.
lParam	Is not used.

Return Value The return value is a data handle that identifies the buffer that holds the contents of the edit control.

Comments An application may send this message to a control only if it has created the dialog box containing the control with the DS_LOCALEDIT style flag set.

EM_GETLINE

This message copies a line from the edit control.

Parameter	Description
wParam	Specifies the line number of the line in the control, where the line number of the first line is zero.
lParam	Points to the buffer where the line will be stored. The first word of the buffer specifies the maximum number of bytes to be copied to the buffer. The copied line is not null-terminated.

Return Value The return value is the number of bytes actually copied. This message is not processed by single-line edit controls.

EM_GETLINECOUNT

This message returns the number of lines of text in the edit control.

Parameter	Description
wParam	Is not used.
lParam	Is not used.

Return Value The return value is the number of lines of text in the control.

Comments This message is not processed by single-line edit controls.

EM_GETMODIFY

This message returns the current value of the modify flag for a given edit control. The flag is set by the control if the user enters or modifies text within the control.

Parameter	Description
wParam	Is not used.
lParam	Is not used.

Return Value The return value is the value of the current modify flag for a given edit control.

EM_GETRECT

This message retrieves the formatting rectangle of the control.

Parameter	Description
wParam	Is not used.
lParam	Points to a **RECT** data structure. The control copies the dimensions of the structure.

EM_GETSEL

This message returns the starting and ending character positions of the current selection.

Parameter	Description
wParam	Is not used.
lParam	Is not used.

Return Value The return value is a long value that contains the starting position in the low-order word. It contains the position of the first nonselected character after the end of the selection in the high-order word.

EM_LIMITTEXT

This message limits the length (in bytes) of the text the user may enter.

Parameter	Description
wParam	Specifies the maximum number of bytes that can be entered. If the user attempts to enter more characters, the edit control beeps and does not accept the characters. If the *wParam* parameter is zero, no limit is imposed on the size of the text (until no more memory is available).
lParam	Is not used.

Comments The EM_LIMITTEXT message does not affect text set by the WM_SETTEXT message or the buffer set by the EM_SETHANDLE message.

EM_LINEFROMCHAR

This message returns the line number of the line that contains the character whose position (indexed from the beginning of the text) is specified by the *wParam* parameter.

Parameter	Description
wParam	Contains the index value for the desired character in the text of the edit control (these index values are zero-based), or contains –1.
lParam	Is not used.

Return Value The return value is a line number. If *wParam* is –1, the number of the line that contains the first character of the selection is returned; otherwise, *wParam* contains the index (or position) of the desired character in the edit-control text, and the number of the line that contains that character is returned.

EM_LINEINDEX

This message returns the number of character positions that occur preceding the first character in a given line.

Parameter	Description
wParam	Specifies the desired line number, where the line number of the first line is zero. If the *wParam* parameter is –1, the current line number (the line that contains the caret) is used.
lParam	Is not used.

Return Value The return value is the number of character positions that precede the first character in the line.

Comments This message will not be processed by single-line edit controls.

EM_LINELENGTH

This message returns the length of a line (in bytes) in the edit control's text buffer.

Parameter	Description
wParam	Specifies the character index of a character in the specified line, where the line number of the first line is zero. If the *wParam* parameter is –1, the length of the current line (the line that contains the caret) is returned, not including the length of any selected text. If the current selection spans more than one line, the total length of the lines, minus the length of the selected text, is returned.
lParam	Is not used.

Comments Use the EM_LINEINDEX message to retrieve a character index for a given line number. This index can be used with the EM_LINELENGTH message.

EM_LINESCROLL

This message scrolls the content of the control by the given number of lines.

Parameter	Description
wParam	Is not used.
lParam	Contains the number of lines and character positions to scroll. The low-order word of the *lParam* parameter contains the number of lines to scroll vertically; the high-order word contains the number of character positions to scroll horizontally.

Comments This message will not be processed by single-line edit controls.

EM_REPLACESEL

This message replaces the current selection with new text.

Parameter	Description
wParam	Is not used.
lParam	Points to a null-terminated string of replacement text.

EM_SETHANDLE

This message establishes the text buffer used to hold the contents of the control window.

Parameter	Description
wParam	Contains a handle to the buffer. The handle must be a local handle to a location in the application's data segment. The edit control uses this buffer to store the currently displayed text, instead of allocating its own buffer. If necessary, the control reallocates this buffer.
lParam	Is not used.

Comments This message will not be processed by single-line edit controls.

If the EM_SETHANDLE message is used to change the text buffer used by an edit control, the previous text buffer is not destroyed. The application must retrieve the previous buffer handle before setting the new handle, and must free the old handle by using the **LocalFree** function.

An edit control automatically reallocates the given buffer whenever it needs additional space for text, or it removes enough text so that additional space is no longer needed. An application may send this message to a control only if it has created the dialog box containing the control with the DS_LOCALEDIT style flag set.

EM_SETMODIFY

This message sets the modify flag for a given edit control.

Parameter	Description
wParam	Specifies the new value for the modify flag.
lParam	Is not used.

EM_SETPASSWORDCHAR 3.0

This message sets the character displayed in an edit control created with the ES_PASSWORD style. The default display character is an asterisk (*).

Parameter	Description
wParam	Specifies the character to be displayed in place of the character typed by the user. If *wParam* is NULL, the actual characters typed by the user are displayed.
lParam	Is not used.

EM_SETRECT

This message sets the formatting rectangle for a control. The text is reformatted and redisplayed to reflect the changed rectangle.

Parameter	Description
wParam	Is not used.
lParam	Points to a **RECT** data structure that specifies the new dimensions of the rectangle.

Comments This message will not be processed by single-line edit controls.

EM_SETRECTNP

This message sets the formatting rectangle for a control. The text is reformatted and redisplayed to reflect the changed rectangle. The EM_SETRECTNP message is the same as the EM_SETRECT message, except that the control is *not* repainted. Any subsequent alterations cause the control to be repainted to reflect the changed formatting rectangle. This message is used when the field is to be repainted later.

Parameter	Description
wParam	Is not used.
lParam	Points to a **RECT** data structure that specifies the new dimensions of the rectangle.

Comments This message will not be processed by single-line edit controls.

EM_SETSEL

This message selects all characters in the current text that are within the starting and ending character positions given by the *lParam* parameter.

Parameter	Description
wParam	Is not used.
lParam	Specifies the starting position in the low-order word and the ending position in the high-order word. The position values 0 to 32,767 select the entire string.

EM_SETTABSTOPS 3.0

This message sets the tab-stop positions in a multiline edit control.

Parameter	Description
wParam	Is an integer that specifies the number of tab stops in the edit control.
lParam	Is a long pointer to the first member of an array of integers containing the tab stop positions in dialog units. (A dialog unit is a horizontal or vertical distance. One horizontal dialog unit is equal to ¼ of the current dialog base width unit. The dialog base units are computed based on the height and width of the current system font. The **GetDialogBaseUnits** function returns the current dialog base units in pixels.) The tab stops must be sorted in increasing order; back tabs are not allowed.

Return Value The return value is TRUE if all the tabs were set. Otherwise, the return value is FALSE.

Comments If *wParam* is zero and *lParam* is NULL, the default tab stops are set at every 32 dialog units.

If *wParam* is 1, the edit control will have tab stops separated by the distance specified by *lParam.*

If *lParam* points to more than a single value, then a tab stop will be set for each value in *lParam*, up to the number specified by *wParam.*

EM_SETWORDBREAK

This message is sent to the multiline edit control, informing the edit control that Windows has replaced the default word-break function with an application-supplied word-break function. A word-break function scans a text buffer (which contains text to be sent to the display), looking for the first word that will not fit on the current display line. The word-break function places this word at the beginning of the next line on the display. A word-break function defines at what point Windows should break a line of text for multiline edit controls, usually at a blank character that separates two words. The default word-break function breaks a line of text at a blank character. The application-supplied function may define a word break to be a hyphen or character other than the blank character.

Parameter	Description
wParam	Is not used.
lParam	Is a procedure-instance address.

Comments

The callback-function address, passed as the *lParam* parameter, must be created by using the **MakeProcInstance** function.

The callback function must use the Pascal calling convention and must be declared **FAR**.

Callback Function

LPSTR FAR PASCAL *WordBreakFunc*(*lpchEditText*, *ichCurrentWord*, *cchEditText*)
LPSTR *lpchEditText*;
short *ichCurrentWord*;
short *cchEditText*;

WordBreakFunc is a placeholder for the application-supplied function name. The actual name must be exported by including it in an **EXPORTS** statement in the application's module-definition file.

Parameter	Description
lpchEditText	Points to the text of the edit control.
ichCurrentWord	Specifies an index to a word in the buffer of text that identifies at what point the function should begin checking for a word break.
cchEditText	Specifies the number of bytes of edit text.

Return Value

The return value points to the first byte of the next word in the edit-control text. If the current word is the last word in the text, the return value points to the first byte that follows the last word.

EM_UNDO

This message undoes the last edit to the edit control. When the user modifies the edit control, the last change is stored in an undo buffer, which grows dynamically as required. If insufficient space is available for the buffer, the undo attempt fails and the edit control is unchanged.

Parameter	Description
wParam	Is not used.
lParam	Is not used.

Return Value The return value is nonzero if the undo operation is successful. It is zero if the undo operation fails.

EN_CHANGE

This code specifies that the user has taken an action that may have altered text. It is sent after Windows updates a display (unlike the EN_UPDATE code). The control's parent window receives this code through a WM_COMMAND message from the control.

Parameter	Description
wParam	Contains the *wParam* parameter of the WM_COMMAND message, and specifies the control ID.
lParam	Contains an edit-control window handle in its low-order word and the EN_CHANGE code in its high-order word.

EN_ERRSPACE

This code specifies that the edit control cannot allocate additional memory space. The control's parent window receives this code through a WM_COMMAND message from the control.

Parameter	Description
wParam	Contains the *wParam* parameter of the WM_COMMAND message, and specifies the control ID.
lParam	Contains an edit-control window handle in its low-order word and the EN_ERRSPACE code in its high-order word.

EN_HSCROLL

This code specifies that the user has clicked the edit control's horizontal scroll bar. The control's parent window receives this code through a WM_COMMAND message from the control. The parent window is notified before the screen is updated.

Parameter	Description
wParam	Contains the *wParam* parameter of the WM_COMMAND message, and specifies the control ID.
lParam	Contains an edit-control window handle in its low-order word and the EN_HSCROLL code in its high-order word.

EN_KILLFOCUS

This code specifies that the edit control has lost the input focus. The control's parent window receives this code through a WM_COMMAND message from the control.

Parameter	Description
wParam	Contains the *wParam* parameter of the WM_COMMAND message, and specifies the control ID.
lParam	Contains an edit-control window handle in its low-order word and the EN_KILLFOCUS code in its high-order word.

EN_MAXTEXT 3.0

This code specifies that the current insertion has exceeded the specified number of characters for the edit control. The insertion has been truncated. This message is also sent when an edit control does not have the ES_AUTOHSCROLL style and the number of characters to be inserted would exceed the width of the edit control. The control's parent window receives this code through a WM_COMMAND message from the control.

Parameter	Description
wParam	Contains the *wParam* parameter of the WM_COMMAND message, and specifies the control ID.
lParam	Contains an edit-control window handle in its low-order word and the EN_MAXTEXT code in its high-order word.

EN_SETFOCUS

This code specifies that the edit control has obtained the input focus. The control's parent window receives this code through a WM_COMMAND message from the control.

Parameter	Description
wParam	Contains the *wParam* parameter of the WM_COMMAND message, and specifies the control ID.
lParam	Contains an edit-control window handle in its low-order word and the EN_SETFOCUS code in its high-order word.

EN_UPDATE

The code specifies that the edit control will display altered text. The control's parent window receives this code through a WM_COMMAND message from the control; notification occurs after the control has formatted the text, but before it displays the text. This makes it possible to alter the window size, if necessary.

Parameter	Description
wParam	Specifies the control ID.
lParam	Contains an edit-control window handle in its low-order word and the EN_UPDATE code in its high-order word.

EN_VSCROLL

This code specifies that the user has clicked the edit control's vertical scroll bar. The control's parent window receives this code through a WM_COMMAND message from the control; notification occurs before the screen is updated.

Parameter	Description
wParam	Contains the *wParam* parameter of the WM_COMMAND message, and specifies the control ID.
lParam	Contains an edit-control window handle in its low-order word and the EN_VSCROLL code in its high-order word.

LB_ADDSTRING

This message adds a string to the list box. If the list box is not sorted, the string is added to the end of the list. If the list box is sorted, the string is inserted into the list after sorting.

This message removes any existing list-box selections.

Parameter	Description
wParam	Is not used.
lParam	Points to the null-terminated string that is to be added. If the list box was created with an owner-draw style but without the LBS_HASSTRINGS style, the *lParam* parameter is an application-supplied 32-bit value that is stored by the list box instead of the pointer to the string.

Return Value The return value is the index to the string in the list box. The return value is LB_ERR if an error occurs; the return value is LB_ERRSPACE if insufficient space is available to store the new string.

Comments If an owner-draw list box was created with the LBS_SORT style but not the LBS_HASSTRINGS style, the WM_COMPAREITEM message is sent one or more times to the owner of the list box so the new item can be properly placed in the list box.

LB_DELETESTRING

This message deletes a string from the list box.

Parameter	Description
wParam	Contains an index to the string that is to be deleted.
lParam	Is not used.

Return Value The return value is a count of the strings remaining in the list. The return value is LB_ERR if an error occurs.

Comments If the list box was created with an owner-draw style but without the LBS_HASSTRINGS style, a WM_DELETEITEM message is sent to the owner of the list box so the application can free additional data associated with the item (through the *lParam* parameter of the LB_ADDSTRING or LB_INSERTSTRING message).

LB_DIR

This message adds a list of the files from the current directory to the list box. Only files with the attributes specified by the *wParam* parameter and that match the file specification given by the *lParam* parameter are added.

Parameter	Description
wParam	Contains a DOS attribute value. For a list of the DOS attributes, see the **DlgDirList** function in Chapter 4, "Functions Directory."
lParam	Points to a file-specification string. The string can contain wild-card characters (for example, *.*).

Return Value The return value is a count of items displayed. The return value is LB_ERR if an error occurs; the return value is LB_ERRSPACE if insufficient space is available to store the new strings.

Comments The return value of the LB_DIR message is one less than the return value of the LB_GETCOUNT message.

LB_FINDSTRING 3.0

This message finds the first string in the list box which matches the given prefix text.

Parameter	Description
wParam	Contains the index of the item before the first item to be searched. When the search reaches the bottom of the list box it continues from the top of the list box back to the item specified by *wParam*. If the *wParam* parameter is –1, the entire list box is searched from the beginning.
lParam	Points to the prefix string. The string must be null-terminated.

Return Value The return value is the index of the matching item or LB_ERR if the search was unsuccessful.

Comments If the list box was created with an owner-draw style but without the LBS_HASSTRINGS style, this message returns the index of the item whose long value (supplied as the *lParam* parameter of the LB_ADDSTRING or LB_INSERTSTRING message) matches the value supplied as the *lParam* parameter of LB_FINDSTRING.

LB_GETCOUNT

This message returns a count of the items in the list box.

Parameter	Description
wParam	Is not used.
lParam	Is not used.

Return Value The return value is a count of the items in the list box. The return value is LB_ERR if an error occurs.

LB_GETCURSEL

This message returns the index of the currently selected item, if any.

Parameter	Description
wParam	Is not used.
lParam	Is not used.

Return Value The return value is the index of the currently selected item. It is LB_ERR if no item is selected or if the list-box type is multiple selection.

LB_GETHORIZONTALEXTENT 3.0

This message retrieves from a list box the width in pixels by which the list box can be scrolled horizontally if the list box has horizontal scroll bars.

Parameter	Description
wParam	Is not used.
lParam	Is not used.

Return Value The return value is the scrollable width of the list box, in pixels.

Comments To respond to the LB_GETHORIZONTALEXTENT message, the list box must have been defined with the WS_HSCROLL style.

LB_GETITEMDATA 3.0

This message retrieves the application-supplied 32-bit value associated with the specified list-box item. If the item is in an owner-draw list box created without the LBS_HASSTRINGS style, this 32-bit value was contained in the *lParam* parameter of the LB_ADDSTRING or LB_INSERTSTRING message that added the item to the list box. Otherwise, it was the value in the *lParam* parameter of a LB_SETITEMDATA message.

Parameter	Description
wParam	Contains an index to the item.
lParam	Is not used.

Return Value The return value is the 32-bit value associated with the item, or LB_ERR if an error occurs.

LB_GETITEMRECT 3.0

This message retrieves the dimensions of the rectangle that bounds a list-box item as it is currently displayed in the list-box window.

Parameter	Description
wParam	Contains an index to the item.
lParam	Contains a long pointer to a **RECT** data structure that receives the list-box client coordinates of the item.

Return Value The return value is LB_ERR if an error occurs.

LB_GETSEL

This message returns the selection state of an item.

Parameter	Description
wParam	Contains an index to the item.
lParam	Is not used.

Return Value The return value is a positive number if an item is selected. Otherwise, it is zero. The return value is LB_ERR if an error occurs.

LB_GETSELCOUNT 3.0

This message returns the total number of selected items in a multiselection list box.

Parameter	Description
wParam	Not used.
lParam	Not used.

Return Value The return value is the count of selected items in a list box. If the list box is a single-selection list box, the return value is LB_ERR.

LB_GETSELITEMS 3.0

This message fills a buffer with an array of integers specifying the item numbers of selected items in a multiselection list box.

Parameter	Description
wParam	Specifies the maximum number of selected items whose item numbers are to be placed in the buffer.
lParam	Contains a long pointer to a buffer large enough for the number of integers specified by the *wParam* parameter.

Return Value The return value is the actual number of items placed in the buffer. If the list box is a single-selection list box, the return value is LB_ERR.

LB_GETTEXT

This message copies a string from the list into a buffer.

Parameter	Description
wParam	Contains the index of the string to be copied.

Parameter	Description
lParam	Points to the buffer that is to receive the string. The buffer must have both sufficient space for the string and a terminating null character.

Return Value The return value is the length of the string (in bytes), excluding the terminating null character. The return value is LB_ERR if the *wParam* parameter is not a valid index.

Comments If the list box was created with an owner-draw style but without the LBS_HASSTRINGS style, the buffer pointed to by the *lParam* parameter of the message receives the 32-bit value associated with the item through the *lParam* parameter of the LB_ADDSTRING or LB_INSERTSTRING message.

LB_GETTEXTLEN

This message returns the length of a string in the list box.

Parameter	Description
wParam	Contains an index to the string.
lParam	Is not used.

Return Value The return value is the length of the string (in bytes), excluding the terminating null character. The return value is LB_ERR if an error occurs.

LB_GETTOPINDEX 3.0

This message returns the index of the first visible item in a list box. Initially, item 0 is at the top of the list box, but if the list box is scrolled, another item may be at the top.

Parameter	Description
wParam	Not used.
lParam	Not used.

Return Value The index of the first visible item in a list box.

LB_INSERTSTRING

This message inserts a string into the list box. No sorting is performed.

Parameter	Description
wParam	Contains an index to the position that will receive the string. If the *wParam* parameter is –1, the string is added to the end of the list.
lParam	Points to the null-terminated string that is to be inserted. If the list box was created with an owner-draw style but without the LBS_HASSTRINGS style, the *lParam* parameter is an application-supplied 32-bit value that is stored by the list box instead of the pointer to the string.

Return Value The return value is the index of the position at which the string was inserted. The return value is LB_ERR if an error occurs; the return value is LB_ERRSPACE if insufficient space is available to store the new string.

LB_RESETCONTENT

This message removes all strings from a list box and frees any memory allocated for those strings.

Parameter	Description
wParam	Is not used.
lParam	Is not used.

Comments If the list box was created with an owner-draw style but without the LBS_HASSTRINGS style, the owner of the list box receives a WM_DELETEITEM message for each item in the list box.

LB_SELECTSTRING

This message changes the current selection to the first string that has the specified prefix.

Parameter	Description
wParam	Contains the index of the item before the first item to be searched. When the search reaches the bottom of the list box it continues from the top of the list box back to the item specified by *wParam*. If the *wParam* parameter is –1, the entire list box is searched from the beginning.
lParam	Points to the prefix string. The string must have a null-terminating character.

Return Value The return value is the index of the selected item. The return value is LB_ERR if an error occurs.

Comments This message must not be used with list boxes that are multiple-selection type.

A string is selected only if its initial characters (from the starting point) match the characters in the prefix string.

If the list box was created with an owner-draw style but without the LBS_HASSTRINGS style, this message returns the index of the item whose long value (supplied as the *lParam* parameter of the LB_ADDSTRING or LB_INSERTSTRING message) matches the value supplied as the *lParam* parameter of LB_FINDSTRING.

LB_SELITEMRANGE 3.0

This message selects one or more consecutive items in a multiple-selection list box.

Parameter	Description
wParam	Specifies how to set the selection. If the *wParam* parameter is nonzero, the string is selected and highlighted; if *wParam* is zero, the highlight is removed and the string is no longer selected.
lParam	The low-order word of the *lParam* parameter is an index that specifies the first item to set, and the high-order word is an index that specifies the last item to set.

Return Value The return value is LB_ERR if an error occurs.

Comments This message should be used only with multiple-selection list boxes.

LB_SETCOLUMNWIDTH 3.0

This message is sent to a multicolumn list box created with the LBS_MULTICOLUMN style to set the width in pixels of all columns in the list box.

Parameter	Description
wParam	Specifies the width in pixels of all columns.
lParam	Is not used.

LB_SETCURSEL

This message selects a string and scrolls it into view, if necessary. When the new string is selected, the list box removes the highlight from the previously selected string.

Parameter	Description
wParam	Contains the index of the string that is selected. If *wParam* is –1, the list box is set to have no selection.
lParam	Is not used.

Return Value The return value is LB_ERR if an error occurs.

Comments This message should be used only with single-selection list boxes. It cannot be used to set or remove a selection in a multiple-selection list box.

LB_SETHORIZONTALEXTENT 3.0

This message sets the width in pixels by which a list box can be scrolled horizontally. If the size of the list box is smaller than this value, the horizontal scroll bar will horizontally scroll items in the list box. If the list box is as large or larger than this value, the horizontal scroll bar is disabled.

Parameter	Description
wParam	Specifies the number of pixels by which the list box can be scrolled.
lParam	Is not used.

Comments To respond to the LB_SETHORIZONTALEXTENT message, the list box must have been defined with the WS_HSCROLL style.

LB_SETITEMDATA 3.0

This message sets a 32-bit value associated with the specified item in a list box. If the item is in an owner-draw list box created without the LBS_HASSTRINGS style, this message replaces the 32-bit value that was contained in the *lParam* parameter of the LB_ADD-STRING or LB_INSERTSTRING message that added the item to the list box.

Parameter	Description
wParam	Contains an index to the item.
lParam	Contains the new value to be associated with the item.

Return Value The return value is LB_ERR if an error occurs.

LB_SETSEL

This message selects a string in a multiple-selection list box.

Parameter	Description
wParam	Specifies how to set the selection. If the *wParam* parameter is nonzero, the string is selected and highlighted; if *wParam* is zero, the highlight is removed and the string is no longer selected.
lParam	The low-order word of the *lParam* parameter is an index that specifies which string to set. If *lParam* is –1, the selection is added to or removed from all strings, depending on the value of *wParam.*

Return Value The return value is LB_ERR if an error occurs.

Comments This message should be used only with multiple-selection list boxes.

LB_SETTABSTOPS 3.0

This message sets the tab-stop positions in a list box.

Parameter	Description
wParam	Is an integer that specifies the number of tab stops in the list box.
lParam	Is a long pointer to the first member of an array of integers containing the tab stop positions in dialog units. (A dialog unit is a horizontal or vertical distance. One horizontal dialog unit is equal to ¼ of the the current dialog base width unit. The dialog base units are computed based on the height and width of the current system font. The **GetDialogBaseUnits** function returns the current dialog base units in pixels.) The tab stops must be sorted in increasing order; back tabs are not allowed.

Return Value The return value is TRUE if all the tabs were set. Otherwise, the return value is FALSE.

Comments If *wParam* is zero and *lParam* is NULL, the default tab stop is two dialog units.

If *wParam* is 1, the edit control will have tab stops separated by the distance specified by *lParam*.

If *lParam* points to more than a single value, then a tab stop will be set for each value in *lParam*, up to the number specified by *wParam*.

To respond to the LB_SETTABSTOPS message, the list box must have been created with the LBS_USETABSTOPS style.

LB_SETTOPINDEX 3.0

This message sets the first visible item in a list box to the item identified by the index.

Parameter	Description
wParam	Specifies the index of the list-box item.
lParam	Not used.

Return Value The return value is LB_ERR if an error occurs.

LBN_DBLCLK

This code specifies that the user has double-clicked a string. The control's parent window receives this code through a WM_COMMAND message from the control.

Parameter	Description
wParam	Contains the *wParam* parameter of the WM_COMMAND message, and specifies the control ID.
lParam	Contains an edit-control window handle in its low-order word and the LBN_DBLCLK code in its high-order word.

Comments This code applies only to list-box controls that have LBS_NOTIFY style.

LBN_ERRSPACE

This code specifies that the list-box control cannot allocate enough memory to meet a specific request. The control's parent window receives this code through a WM_COMMAND message from the control.

Parameter	Description
wParam	Contains the *wParam* parameter of the WM_COMMAND message, and specifies the control ID.
lParam	Contains a list-box window handle in its low-order word and the LBN_ERRSPACE code in its high-order word.

Comments This code applies only to list-box controls that have LBS_NOTIFY style.

LBN_KILLFOCUS 3.0

This code is sent when a list box loses input focus. The control's parent window receives this code through a WM_COMMAND message from the control.

Parameter	Description
wParam	Specifies the control ID of the list box.
lParam	Contains the list-box window handle in its low-order word and the LBN_KILLFOCUS code in its high-order word.

LBN_SELCHANGE

This code specifies that the selection in a list box has changed. The control's parent window receives this code through a WM_COMMAND message from the control.

Parameter	Description
wParam	Contains the *wParam* parameter of the WM_COMMAND message, and specifies the control ID.
lParam	Contains a list-box window handle in its low-order word and the LBN_SELCHANGE code in its high-order word.

Comments This code applies only to list-box controls that have LBS_NOTIFY style.

LBN_SETFOCUS 3.0

This code is sent when the list box receives input focus. The control's parent window receives this code through a WM_COMMAND message from the control.

Parameter	Description
wParam	Specifies the control ID of the list box.
lParam	Contains the list-box window handle in its low-order word and the LBN_SETFOCUS code in its high-order word.

WM_ACTIVATE

This message is sent when a window becomes active or inactive.

Parameter	Description
wParam	Specifies the new state of the window. The *wParam* parameter is zero if the window is inactive; it is one of the following nonzero values if the window is being activated:

Value	Meaning
1	The window is being activated through some method other than a mouse click (for example, through a call to the **SetActiveWindow** function or selection of the window by the user through the keyboard interface).
2	The window is being activated by a mouse click by the user. Any mouse button can be clicked: right, left, or middle.

Parameter	Description
lParam	Identifies a window and specifies its state. The high-order word of the *lParam* parameter is nonzero if the window is minimized. Otherwise, it is zero. The value of the low-order word of *lParam* depends on the value of the *wParam* parameter. If *wParam* is zero, the low-order word of *lParam* is a handle to the window being activated. If *wParam* is nonzero, the low-order word of *lParam* is the handle of the window being inactivated (this handle may be NULL).

Default Action If the window is being activated and is not minimized, the **DefWindowProc** function sets the input focus to the window.

WM_ACTIVATEAPP

This message is sent when a window being activated belongs to a different application than the currently active window. The message is sent to the application whose window will be activated and the application whose window will be deactivated.

Parameter	Description
wParam	Specifies whether a window is being activated or deactivated. A nonzero value indicates that Windows will activate a window; zero indicates that Windows will deactivate a window.
lParam	Contains the task handle of the application. If the *wParam* parameter is zero, the low-order word of the *lParam* parameter contains the task handle of the application that owns the window that is being deactivated. If *wParam* is nonzero, the low-order word of *lParam* contains the task handle of the application that owns the window that is being activated. The high-order word is not used.

WM_ASKCBFORMATNAME

This message is sent when the clipboard contains a data handle for the CF_OWNERDISPLAY format (that is, the clipboard owner should display the clipboard contents), and requests a copy of the format name.

Parameter	Description
wParam	Specifies the maximum number of bytes to copy.
lParam	Points to the buffer where the copy of the format name is to be stored.

Comments The clipboard owner should copy the name of the CF_OWNERDISPLAY format into the specified buffer, not exceeding the maximum number of bytes.

WM_CANCELMODE

This message cancels any mode the system is in, such as one that tracks the mouse in a scroll bar or moves a window. Windows sends the WM_CANCELMODE message when an application displays a message box.

Parameter	Description
wParam	Is not used.
lParam	Is not used.

WM_CHANGECBCHAIN

This message notifies the first window in the clipboard-viewer chain that a window is being removed from the chain.

Parameter	Description
wParam	Contains the handle to the window that is being removed from the clipboard-viewer chain.
lParam	Contains in its low-order word the handle to the window that follows the window being removed from the clipboard-viewer chain.

Comments Each window that receives the WM_CHANGECBCHAIN message should call the **SendMessage** function to pass on the message to the next window in the clipboard-viewer chain. If the window being removed is the next window in the chain, the window specified by the low-order word of the *lParam* parameter becomes the next window, and clipboard messages are passed on to it.

WM_CHAR

This message results when a WM_KEYUP and a WM_KEYDOWN message are translated. It contains the value of the keyboard key being pressed or released.

Parameter	Description
wParam	Contains the value of the key.
lParam	Contains the repeat count, scan code, key-transition code, previous key state, and context code, as shown in the following list:

Bit	Value
0–15 (low-order word)	Repeat count (the number of times the key stroke is repeated as a result of the user holding down the key).
16–23 (low byte of high-order word)	Scan code (OEM-dependent value).
24	Extended key, such as a function key or a key on the numeric keypad (1 if it is an extended key).

Parameter	Description	
	Bit	**Value**
	25–26	Not used.
	27–28	Used internally by Windows.
	29	Context code (1 if the ALT key is held down while the key is pressed, 0 otherwise).
	30	Previous key state (1 if the key is down before the message is sent, 0 if the key is up).
	31	Transition state (1 if the key is being released, 0 if the key is being pressed).

Comments

Since there is not necessarily a one-to-one correspondence between keys pressed and character messages generated, the information in the high-order word of the *lParam* parameter is generally not useful to applications. The information in the high-order word applies only to the most recent WM_KEYUP or WM_KEYDOWN message that precedes the posting of the character message.

For IBM® Enhanced 101- and 102-key keyboards, enhanced keys are the right ALT and the right CONTROL keys on the main section of the keyboard; the INSERT, DELETE, HOME, END, PAGE UP, PAGE DOWN and DIRECTION keys in the clusters to the left of the numeric key pad; and the divide (/) and ENTER keys in the numeric key pad. Some other keyboards may support the extended-key bit in the *lParam* parameter.

WM_CHARTOITEM 3.0

This message is sent by a list box with the LBS_WANTKEYBOARDINPUT style to its owner in response to a WM_CHAR message.

Parameter	Description
wParam	Contains the value of the key which the user pressed.
lParam	Contains the current caret position in its high-order word and the window handle of the list box in its low-order word.

Return Value The return value specifies the action which the application performed in response to the message. A return value of –2 indicates that the application handled all apsects of selecting the item and wants no further action by the list box. A return value of –1 indicates that the list box should perform the default action in response to the key stroke. A return value of zero or greater specifies the index of an item in the list box and indicates that the list box should perform the default action for the key stroke on the given item.

WM_CHILDACTIVATE

This message is sent to a child window's parent window when the **SetWindowPos** function moves a child window.

Parameter	Description
wParam	Is not used.
lParam	Is not used.

WM_CLEAR

This message deletes the current selection.

Parameter	Description
wParam	Is not used.
lParam	Is not used.

WM_CLOSE

This message occurs when a window is closed.

Parameter	Description
wParam	Is not used.
lParam	Is not used.

Default Action The **DefWindowProc** function calls the **DestroyWindow** function to destroy the window.

Comments An application can prompt the user for confirmation, prior to destroying a window, by processing the WM_CLOSE message and calling the **DestroyWindow** function only if the user confirms the choice.

WM_COMMAND

This message occurs when the user selects an item from a menu, when a control passes a message to its parent window, or when an accelerator key stroke is translated.

Parameter	Description
wParam	Contains the menu item, the control ID, or the accelerator ID.
lParam	Specifies whether the message is from a menu, an accelerator, or a control. The low-order word contains zero if the message is from a menu. The high-order word contains 1 if the message is an accelerator message. If the message is from a control, the high-order word of the *lParam* parameter contains the notification code. The low-order word is the window handle of the control sending the message.

Comments Accelerator key strokes that are defined to select items from the System menu are translated into WM_SYSCOMMAND messages.

If an accelerator key stroke that corresponds to a menu item occurs when the window that owns the menu is minimized, no WM_COMMAND message is sent. However, if an accelerator key stroke that does not match any of the items on the window's menu or on the System menu occurs, a WM_COMMAND message is sent, even if the window is minimized.

WM_COMPACTING 3.0

This message is sent to all top-level windows when Windows detects that more than 12.5 percent of system time over a 30- to 60-second interval is being spent compacting memory. This indicates that system memory is low.

When an application receives this message, it should free as much memory as possible, taking into account the current level of activity of the application and the total number of applications running in Windows. The application can call the **GetNumTasks** function to determine how many applications are running.

Parameter	Description
wParam	Specifies the ratio of CPU time currently spent by Windows compacting memory. For example, 8000h represents 50% of CPU time.
lParam	Is not used.

WM_COMPAREITEM 3.0

This message determines the relative position of a new item in a sorted owner-draw combo or list box.

Whenever the application adds a new item, Windows sends this message to the owner of a combo or list box created with the CBS_SORT or LBS_SORT style. The *lParam* parameter of the message is a long pointer to a **COMPAREITEMSTRUCT** data structure that contains the identifiers and application-supplied data for two items in the combo or list box. When the owner receives the message, the owner returns a value indicating which of the items should appear before the other. Typically, Windows sends this message several times until it determines the exact position for the new item.

Parameter	Description
wParam	Is not used.
lParam	Contains a long pointer to a **COMPAREITEMSTRUCT** data structure that contains the identifiers and application-supplied data for two items in the combo or list box.

Return Value The return value indicates the relative position of the two items. It may be any of the following values:

Value	Meaning
–1	Item 1 sorts before item 2.
0	Item 1 and item 2 sort the same.
1	Item 1 sorts after item 2.

WM_COPY

This message sends the current selection to the clipboard in CF_TEXT format.

Parameter	Description
wParam	Is not used.
lParam	Is not used.

WM_CREATE

This message informs the window procedure that it can perform any initialization. The **CreateWindow** function sends this message before it returns and before the window is opened.

Parameter	Description
wParam	Is not used.
lParam	Points to a **CREATESTRUCT** data structure that contains copies of parameters passed to the **CreateWindow** function.

WM_CTLCOLOR

This message is sent to the parent window of a predefined control or message box when the control or message box is about to be drawn. By responding to this message, the parent window can set the text and background colors of the child window by using the display-context handle given in the *wParam* parameter.

Parameter	Description
wParam	Contains a handle to the display context for the child window.
lParam	The low-order word of the *lParam* parameter contains the handle to the child window. The high-order word is one of the following values, specifying the type of control:

Value	Control Type
CTLCOLOR_BTN	Button control
CTLCOLOR_DLG	Dialog box

Parameter	Description	
	Value	**Control Type**
	CTLCOLOR_EDIT	Edit control
	CTLCOLOR_LISTBOX	List-box control
	CTLCOLOR_MSGBOX	Message box
	CTLCOLOR_SCROLLBAR	Scroll-bar control
	CTLCOLOR_STATIC	Static control

Default Action

The **DefWindowProc** function selects the default system colors.

Comments

When processing the WM_CTLCOLOR message, the application must align the origin of the intended brush with the window coordinates by first calling the **UnrealizeObject** function for the brush, and then setting the brush origin to the upper-left corner of the window.

If an application processes the WM_CTLCOLOR message, it must return a handle to the brush that is to be used for painting the control background. Note that failure to return a valid brush handle will place the system in an unstable state.

WM_CUT

This message sends the current selection to the clipboard in CF_TEXT format, and then deletes the selection from the control window.

Parameter	Description
wParam	Is not used.
lParam	Is not used.

WM_DEADCHAR

This message results when a WM_KEYUP and a WM_KEYDOWN message are translated. It specifies the character value of a dead key. A dead key is a key, such as the umlaut (double-dot) character, that is combined with other characters to form a composite character. For example, the umlaut-O character consists of the dead key, umlaut, and the O key.

Parameter	Description
wParam	Contains the dead-key character value.
lParam	Contains the repeat count, scan code, key-transition code, previous key state, and context code, as shown in the following list:

Bit	Value
0–15 (low-order word)	Repeat count (the number of times the key stroke is repeated as a result of the user holding down the key).
16–23 (low byte of high-order word)	Scan code (OEM-dependent value).
24	Extended key, such as a function key or a key on the numeric keypad (1 if it is an extended key, 0 otherwise).
25–26	Not used.
27–28	Used internally by Windows.
29	Context code (1 if the ALT key is held down while the key is pressed, 0 otherwise).
30	Previous key state (1 if the key is down before the message is sent, 0 if the key is up).
31	Transition state (1 if the key is being released, 0 if the key is being pressed).

Comments

The WM_DEADCHAR message typically is used by applications to give the user feedback about each key pressed. For example, an application can display the accent in the current character position without moving the caret.

Since there is not necessarily a one-to-one correspondence between keys pressed and character messages generated, the information in the high-order word of the *lParam* parameter is generally not useful to applications. The information in the high-order word applies only to the most recent WM_KEYUP or WM_KEYDOWN message that precedes the posting of the character message.

For IBM Enhanced 101- and 102-key keyboards, enhanced keys are the right ALT and the right CONTROL keys on the main section of the keyboard; the INSERT, DELETE, HOME, END, PAGE UP, PAGE DOWN and DIRECTION keys in the clusters to the left of the numeric key

pad; and the divide (/) and ENTER keys in the numeric key pad. Some other keyboards may support the extended-key bit in the *lParam* parameter.

WM_DELETEITEM 3.0

This message informs the owner of an owner-draw list box or combo box that a list-box item has been removed. This message is sent when the list box or combo box is destroyed or the item is removed by the LB_DELETESTRING, LB_RESETCONTENT, CB_DELETESTRING or CB_RESETCONTENT message.

Parameter	Description
wParam	Not used.
lParam	Contains a long pointer to a **DELETEITEMSTRUCT** data structure that contains information about the deleted list-box item.

WM_DESTROY

This message informs the window that it is being destroyed. The **DestroyWindow** function sends the WM_DESTROY message to the window after removing the window from the screen. The WM_DESTROY message is sent to a parent window before any of its child windows are destroyed.

Parameter	Description
wParam	Is not used.
lParam	Is not used.

Comments If the window being destroyed is part of the clipboard-viewer chain (set by using the **SetClipboardViewer** function), the window must remove itself from the clipboard viewer chain by processing the **ChangeClipboardChain** function before returning from the WM_DESTROY message.

WM_DESTROYCLIPBOARD

This message is sent to the clipboard owner when the clipboard is emptied through a call to the **EmptyClipboard** function.

Parameter	Description
wParam	Is not used.
lParam	Is not used.

WM_DEVMODECHANGE

This message is sent to all top-level windows when the user changes device-mode settings.

Parameter	Description
wParam	Is not used.
lParam	Points to the device name specified in the Windows initialization file, WIN.INI.

WM_DRAWCLIPBOARD

This message is sent to the first window in the clipboard-viewer chain when the contents of the clipboard change. Only applications that have joined the clipboard-viewer chain by calling the **SetClipboardViewer** function need to process this message.

Parameter	Description
wParam	Is not used.
lParam	Is not used.

Comments Each window that receives the WM_DRAWCLIPBOARD message should call the **SendMessage** function to pass the message on to the next window in the clipboard-viewer chain. The handle of the next window is returned by the **SetClipboardViewer** function; it may be modified in response to a WM_CHANGECBCHAIN message.

WM_DRAWITEM 3.0

This message informs the owner-draw button, combo box, list box, or menu that a visual aspect of the control has changed. The **itemAction** field in the **DRAWITEMSTRUCT** structure defines the drawing operation that is to be performed. The data in this field allows the control owner to determine what drawing action is required.

Parameter	Description
wParam	Is not used.
lParam	Contains a long pointer to a **DRAWITEMSTRUCT** data structure that contains information about the item to be drawn and the type of drawing required.

Comments Before returning from processing this message, an application should restore all objects selected for the display context supplied in the **hDC** field of the **DRAWITEMSTRUCT** data structure.

WM_ENABLE

This message is sent after a window has been enabled or disabled.

Parameter	Description
wParam	Specifies whether the window has been enabled or disabled. The *wParam* parameter is nonzero if the window has been enabled; it is zero if the window has been disabled.
lParam	Is not used.

WM_ENDSESSION

This message is sent to tell an application that has responded nonzero to a WM_QUERY-ENDSESSION message whether the session is actually being ended.

Parameter	Description
wParam	Specifies whether or not the session is being ended. It is nonzero if the session is being ended. Otherwise, it is zero.
lParam	Is not used.

Comments If the *wParam* parameter is nonzero, Windows can terminate any time after all applications have returned from processing this message. Consequently, an application should perform all tasks required for termination before returning from this message.

The application does not need to call the **DestroyWindow** or **PostQuitMessage** function when the session is being ended.

WM_ENTERIDLE

This message informs an application's main windows procedure that a modal dialog box or a menu is entering an idle state. A modal dialog box or menu enters an idle state when no messages are waiting in its queue after it has processed one or more previous messages.

Parameter	Description
wParam	Specifies whether the message is the result of a dialog box or a menu being displayed. It is one of these values:

Value	Meaning
MSGF_DIALOGBOX	The system is idle because a dialog box is being displayed.
MSGF_MENU	The system is idle because a menu is being displayed.

Parameter	Description
lParam	Contains in its low-order word the handle of the dialog box (if *wParam* is MSGF_DIALOGBOX) or of the window containing the displayed menu (if *wParam* is MSGF_MENU). The high-order word is not used.

Default Action The **DefWindowProc** function returns zero.

WM_ERASEBKGND

This message is sent when the window background needs erasing (for example, when a window is resized). It is sent to prepare an invalidated region for painting.

Parameter	Description
wParam	Contains the device-context handle.
lParam	Is not used.

Return Value The return value is nonzero if the background is erased. Otherwise, it is zero. If the application processes the WM_ERASEBKGND message, it should return the appropriate value.

Default Action The background is erased, using the class background brush specified by the **hbrbackground** field in the class structure.

Comments

If **hbrbackground** is NULL, the application should process the WM_ERASEBKGND message and erase the background color. When processing the WM_ERASEBKGND message, the application must align the origin of the intended brush with the window coordinates by first calling the **UnrealizeObject** function for the brush, and then selecting the brush.

Windows assumes the background should be computed by using the MM_TEXT mapping mode. If the device context is using any other mapping mode, the area erased may not be within the visible part of the client area.

WM_FONTCHANGE

This message occurs when the pool of font resources changes. Any application that adds or removes fonts from the system (for example, through the **AddFontResource** or **RemoveFontResource** function) should send this message to all top-level windows.

Parameter	Description
wParam	Is not used.
lParam	Is not used.

Comments

To send the WM_FONTCHANGE message to all top-level windows, an application can call the **SendMessage** function with the *hWnd* parameter set to 0xFFFF.

WM_GETDLGCODE

This message is sent by Windows to an input procedure associated with a control. Normally, Windows handles all DIRECTION-key and TAB-key input to the control. By responding to the WM_GETDLGCODE message, an application can take control of a particular type of input and process the input itself.

Parameter	Description
wParam	Is not used.
lParam	Is not used.

Return Value

The return value is one or more of the following values, indicating which type of input the application processes:

Value	Meaning
DLGC_DEFPUSHBUTTON	Default push button.
DLGC_HASSETSEL	EM_SETSEL messages.
DLGC_PUSHBUTTON	Push button.
DLGC_RADIOBUTTON	Radio button.
DLGC_WANTALLKEYS	All keyboard input.
DLGC_WANTARROWS	DIRECTION keys.
DLGC_WANTCHARS	WM_CHAR messages.
DLGC_WANTMESSAGE	All keyboard input (the application passes this message on to control).
DLGC_WANTTAB	TAB key.

Default Action The **DefWindowProc** function returns zero.

Comments Although the **DefWindowProc** function always returns zero in response to the WM_GETDLGCODE message, the window functions for the predefined control classes return a code appropriate for each class.

The WM_GETDLGCODE message and the returned values are useful only with user-defined dialog controls or standard controls modified by subclassing.

WM_GETFONT 3.0

This message retrieves from a control the font with which the control is currently drawing its text.

Parameter	Description
wParam	Not used.
lParam	Not used.

Return Value The return value is the handle of the font used by the control, or NULL if it is using the system font.

WM_GETMINMAXINFO

This message is sent to a window whenever Windows needs to know the maximized size of the window, the minimum or maximum tracking size of the window, or the maximized position of the window. The maximized size of a window is the size of a window when its borders are fully extended. The maximum tracking size of a window is the largest window size that can be achieved by using the borders to size the window. The minimum tracking size of a window is the smallest window size that can be achieved by using the borders to size the window.

Parameter	Description
wParam	Is not used.
lParam	Points to an array of five points that contains the following information:

Point	Description
rgpt[0]	Used internally by Windows.
rgpt[1]	The maximized size, which is the screen size by default. The width is (SM_CXSCREEN + (2 × SM_CXFRAME)). The height is (SM_CYSCREEN + (2 × SM_CY-FRAME)).
rgpt[2]	The maximized position of the upper-left corner of the window (in screen coordinates). The default *x* value is SM_CXFRAME. The default *y* value is SM_CYFRAME.
rgpt[3]	The minimum tracking size, which is the iconic size by default. The width is SM_CXMINTRACK. The height is SM_CYMINTRACK.
rgpt[4]	The maximum tracking size, which is less than the screen size by default. The width is (SM_CXSCREEN + (2 × SM_CXFRAME)). The height is (SM_CYSCREEN + (2 × SM_CY-FRAME)).

Comments The array contains default values for each point before Windows sends the WM_GET-MINMAXINFO message. This message gives the application the opportunity to alter the default values.

WM_GETTEXT

This message is used to copy the text that corresponds to a window. For edit controls and combo-box edit controls, the text to be copied is the content of the edit control. For button controls, the text is the button name. For lixt boxes, the text is the currently selected item. For other windows, the text is the window caption.

Parameter	Description
wParam	Specifies the maximum number of bytes to be copied, including the null-terminating character.
lParam	Points to the buffer that is to receive the text.

Return Value The return value is the number of bytes copied. It is LB_ERR if no item is selected or CB_ERR if the combo box has no edit control.

WM_GETTEXTLENGTH

This message is used to find the length (in bytes) of the text associated with a window. The length does not include the null-terminating character. For edit controls and combo-box edit controls, the text is the content of the control. For lixt boxes, the text is the currently selected item. For button controls, the text is the button name. For other windows, the text is the window caption.

Parameter	Description
wParam	Is not used.
lParam	Is not used.

Comments The return value is the length of the given text.

WM_HSCROLL

This message is sent when the user clicks the horizontal scroll bar.

Parameter	Description
wParam	Contains a scroll-bar code that specifies the user's scrolling request. It can be any one of the following values:

Value	Meaning
SB_BOTTOM	Scroll to lower right.
SB_ENDSCROLL	End scroll.
SB_LINEDOWN	Scroll one line down.
SB_LINEUP	Scroll one line up.
SB_PAGEDOWN	Scroll one page down.
SB_PAGEUP	Scroll one page up.
SB_THUMBPOSITION	Scroll to absolute position. The current position is provided in the low-order word of *lParam*.
SB_THUMBTRACK	Drag thumb to specified position. The current position is provided in the low-order word of *lParam*.
SB_TOP	Scroll to upper left.

Parameter	Description
lParam	Specifies the window handle of the control. If the message is sent by a scroll-bar control, the high-order word of the *lParam* parameter contains the window handle of the control. If the message is sent as a result of the user clicking a pop-up window's scroll bar, the high-order word is not used.

Comments

The SB_THUMBTRACK message typically is used by applications that give some feedback while the thumb is being dragged.

If an application scrolls the document in the window, it must also reset the position of the thumb by using the **SetScrollPos** function.

WM_HSCROLLCLIPBOARD

This message is sent when the clipboard contains a data handle for the CF_OWNERDISPLAY format (specifically the clipboard owner should display the clipboard contents) and an event occurs in the clipboard application's horizontal scroll bar.

Parameter	Description
wParam	Contains a handle to the clipboard application window.
lParam	Contains one of the following scroll-bar codes in the low-order word:

Value	Meaning
SB_BOTTOM	Scroll to lower right.
SB_ENDSCROLL	End scroll.
SB_LINEDOWN	Scroll one line down.
SB_LINEUP	Scroll one line up.
SB_PAGEDOWN	Scroll one page down.
SB_PAGEUP	Scroll one page up.
SB_THUMBPOSITION	Scroll to absolute position.
SB_TOP	Scroll to upper left.

The high-order word of the *lParam* parameter contains the thumb position if the scroll-bar code is SB_THUMBPOSITION. Otherwise, the high-order word is not used.

Comments The clipboard owner should use the **InvalidateRect** function or repaint as desired. The scroll-bar position should also be reset.

WM_ICONERASEBKGND 3.0

This message is sent to a minimized (iconic) window when the background of the icon must be filled before painting the icon. A window receives this message only if a class icon is defined for the window. Otherwise, WM_ERASEBKGND is sent instead. Passing this message to the **DefWindowProc** function permits Windows to fill the icon background with the background brush of the parent window.

Parameter	Description
wParam	Contains the device-context handle of the icon.
lParam	Is not used.

WM_INITDIALOG

This message is sent immediately before a dialog box is displayed. By processing this message, an application can perform any initialization before the dialog box is made visible.

Parameter	Description
wParam	Identifies the first control item in the dialog box that can be given the input focus. Generally, this is the first item in the dialog box with WS_TABSTOP style.
lParam	Is the value passed as the *dwInitParam* parameter of the function if the dialog box was created by any of the following functions: ■ **CreateDialogIndirectParam** ■ **CreateDialogParam** ■ **DialogBoxIndirectParam** ■ **DialogBoxParam** Otherwise, *lParam* is not used.

Comments If the application returns a nonzero value in response to the WM_INITDIALOG message, Windows sets the input focus to the item identified by the handle in the *wParam* parameter. The application can return FALSE only if it has set the input focus to one of the controls of the dialog box.

WM_INITMENU

This message is a request to initialize a menu. It occurs when a user moves the mouse into a menu bar and clicks, or presses a menu key. Windows sends this message before displaying the menu. This allows the application to change the state of menu items before the menu is shown.

Parameter	Description
wParam	Contains the menu handle of the menu that is to be initialized.
lParam	Is not used.

Comments A WM_INITMENU message is sent only when a menu is first accessed; only one WM_INITMENU message is generated for each access. This means, for example, that moving the mouse across several menu items while holding down the button does not generate new messages. This message does not provide information about menu items.

WM_INITMENUPOPUP

This message is sent immediately before a pop-up menu is displayed. Processing this message allows an application to change the state of items on the pop-up menu before the menu is shown, without changing the state of the entire menu.

Parameter	Description
wParam	Contains the menu handle of the pop-up menu.
lParam	Specifies the index of the pop-up menu. The low-order word contains the index of the pop-up menu in the main menu. The high-order word is nonzero if the pop-up menu is the system menu. Otherwise, it is zero.

WM_KEYDOWN

This message is sent when a nonsystem key is pressed. A nonsystem key is a keyboard key that is pressed when the ALT key is *not* pressed, or a keyboard key that is pressed when a window has the input focus.

Parameter	Description
wParam	Specifies the virtual-key code of the given key.
lParam	Contains the repeat count, scan code, key-transition code, previous key state, and context code, as shown in the following list:

Parameter	Description	
	Bit	**Value**
	0–15 (low-order word)	Repeat count (the number of times the key stroke is repeated as a result of the user holding down the key).
	16–23 (low byte of high-order word)	Scan code (OEM-dependent value).
	24	Extended key, such as a function key or a key on the numeric key pad (1 if it is an extended key).
	25–26	Not used.
	27–28	Used internally by Windows.
	29	Context code (1 if the ALT key is held down while the key is pressed, 0 otherwise).
	30	Previous key state (1 if the key is down before the message is sent, 0 if the key is up).
	31	Transition state (1 if the key is being released, 0 if the key is being pressed).

For WM_KEYDOWN messages, the key-transition bit (bit 31) is 0 and the context-code bit (bit 29) is 0.

Comments

Because of auto-repeat, more than one WM_KEYDOWN message may occur before a WM_KEYUP message is sent. The previous key state (bit 30) can be used to determine whether the WM_KEYDOWN message indicates the first down transition or a repeated down transition.

For IBM Enhanced 101- and 102-key keyboards, enhanced keys are the right ALT and the right CONTROL keys on the main section of the keyboard; the INSERT, DELETE, HOME, END, PAGE UP, PAGE DOWN and DIRECTION keys in the clusters to the left of the numeric key pad; and the divide (/) and ENTER keys in the numeric key pad. Some other keyboards may support the extended-key bit in the *lParam* parameter.

WM_KEYUP

This message is sent when a nonsystem key is released. A nonsystem key is a keyboard key that is pressed when the ALT key is *not* pressed, or a keyboard key that is pressed when a window has the input focus.

Parameter	Description
wParam	Specifies the virtual-key code of the given key.
lParam	Contains the repeat count, scan code, key-transition code, previous key state, and context code, as shown in the following list:

Bit	Value
0–15 (low-order word)	Repeat count (the number of times the key stroke is repeated as a result of the user holding down the key).
16–23 (low byte of high-order word)	Scan code (OEM-dependent value).
24	Extended key, such as a function key or a key on the numeric key pad (1 if it is an extended key).
25–26	Not used.
27–28	Used internally by Windows.
29	Context code (1 if the ALT key is held down while the key is pressed, 0 otherwise).
30	Previous key state (1 if the key is down before the message is sent, 0 if the key is up).
31	Transition state (1 if the key is being released, 0 if the key is being pressed).

For WM_KEYUP messages, the key-transition bit (bit 31) is 1 and the context-code bit (bit 29) is 0.

Comments

For IBM Enhanced 101- and 102-key keyboards, enhanced keys are the right ALT and the right CONTROL keys on the main section of the keyboard; the INSERT, DELETE, HOME, END, PAGE UP, PAGE DOWN and DIRECTION keys in the clusters to the left of the numeric key

pad; and the divide (/) and ENTER keys in the numeric key pad. Some other keyboards may support the extended-key bit in the *lParam* parameter.

WM_KILLFOCUS

This message is sent immediately before a window loses the input focus.

Parameter	Description
wParam	Contains the handle of the window that receives the input focus (may be NULL).
lParam	Is not used.

Comments If an application is displaying a caret, the caret should be destroyed at this point.

WM_LBUTTONDBLCLK

This message occurs when the user double-clicks the left mouse button.

Parameter	Description
wParam	Contains a value that indicates whether various virtual keys are down. It can be any combination of the following values:

Value	Meaning
MK_CONTROL	Set if CONTROL key is down.
MK_LBUTTON	Set if left button is down.
MK_MBUTTON	Set if middle button is down.
MK_RBUTTON	Set if right button is down.
MK_SHIFT	Set if SHIFT key is down.

Parameter	Description
lParam	Contains the *x*- and *y*-coordinates of the cursor. The *x*-coordinate is in the low-order word; the *y*-coordinate is in the high-order word. These coordinates are always relative to the upper-left corner of the window.

Comments Only windows whose window class has CS_DBLCLKS style can receive double-click messages. Windows generates a double-click message when the user presses, releases, and then presses a mouse button again within the system's double-click time limit. Double-clicking actually generates four messages: a down-click message, an up-click message, the double-click message, and another up-click message.

WM_LBUTTONDOWN

This message occurs when the user presses the left mouse button.

Parameter	Description
wParam	Contains a value that indicates whether various virtual keys are down. It can be any combination of the following values:

Value	Meaning
MK_CONTROL	Set if CONTROL key is down.
MK_MBUTTON	Set if middle button is down.
MK_RBUTTON	Set if right button is down.
MK_SHIFT	Set if SHIFT key is down.

Parameter	Description
lParam	Contains the *x*- and *y*-coordinates of the cursor. The *x*-coordinate is in the low-order word; the *y*-coordinate is in the high-order word. These coordinates are always relative to the upper-left corner of the window.

WM_LBUTTONUP

This message occurs when the user releases the left mouse button.

Parameter	Description
wParam	Contains a value that indicates whether various virtual keys are down. It can be any combination of the following values:

Value	Meaning
MK_CONTROL	Set if CONTROL key is down.
MK_MBUTTON	Set if middle button is down.
MK_RBUTTON	Set if right button is down.

Parameter	Description	
	Value	**Meaning**
	MK_SHIFT	Set if SHIFT key is down.
lParam	Contains the *x*- and *y*-coordinates of the cursor. The *x*-coordinate is in the low-order word; the *y*-coordinate is in the high-order word. These coordinates are always relative to the upper-left corner of the window.	

WM_MBUTTONDBLCLK

This message occurs when the user double-clicks the middle mouse button.

Parameter	Description	
wParam	Contains a value that indicates whether various virtual keys are down. It can be any combination of the following values:	
	Value	**Meaning**
	MK_CONTROL	Set if CONTROL key is down.
	MK_LBUTTON	Set if left button is down.
	MK_MBUTTON	Set if middle button is down.
	MK_RBUTTON	Set if right button is down.
	MK_SHIFT	Set if SHIFT key is down.
lParam	Contains the *x*- and *y*-coordinates of the cursor. The *x*-coordinate is in the low-order word; the *y*-coordinate is in the high-order word. These coordinates are always relative to the upper-left corner of the window.	

Comments

Only windows whose window class has CS_DBLCLKS style can receive double-click messages. Windows generates a double-click message when the user presses, releases, and then presses a mouse button again within the system's double-click time limit. Double-clicking actually generates four messages: a down-click message, an up-click message, the double-click message, and another up-click message.

WM_MBUTTONDOWN

This message occurs when the user presses the middle mouse button.

Parameter	Description
wParam	Contains a value that indicates whether various virtual keys are down. It can be any combination of the following values:

Value	Meaning
MK_CONTROL	Set if CONTROL key is down.
MK_LBUTTON	Set if left button is down.
MK_RBUTTON	Set if right button is down.
MK_SHIFT	Set if SHIFT key is down.

lParam	Contains the *x*- and *y*-coordinates of the cursor. The *x*-coordinate is in the low-order word; the *y*-coordinate is in the high-order word. These coordinates are always relative to the upper-left corner of the window.

WM_MBUTTONUP

This message occurs when the user releases the middle mouse button.

Parameter	Description
wParam	Contains a value that indicates whether various virtual keys are down. It can be any combination of the following values:

Value	Meaning
MK_CONTROL	Set if CONTROL key is down.
MK_LBUTTON	Set if left button is down.
MK_RBUTTON	Set if right button is down.
MK_SHIFT	Set if SHIFT key is down.

lParam	Contains the *x*- and *y*-coordinates of the cursor. The *x*-coordinate is in the low-order word; the *y*-coordinate is in the high-order word. These coordinates are always relative to the upper-left corner of the window.

WM_MDIACTIVATE 3.0

An application sends this message to a multiple document interface (MDI) client window to instruct the client window to activate a different MDI child window. As the client window processes this message, it sends WM_MDIACTIVATE to the child window being deactivated and to the child window being activated.

Parameter	Description
wParam	When the application sends the WM_MDIACTIVATE message to its MDI client window, the *wParam* parameter contains the window handle of the MDI child window to be activated. When the client window sends the message to a child window, *wParam* is TRUE if the child is being activated and FALSE if it is being deactivated.
lParam	When received by an MDI child window, the *lParam* parameter contains in its high-order word the window handle of the child window being deactivated and in its low-order word the window handle of the child window being activated. When this message is sent to the client window, *lParam* should be set to NULL.

Comments MDI child windows are activated independently of the MDI frame window. When the frame becomes active, the child window that was last activated with the WM_MDIACTIVE message receives the WM_NCACTIVATE message to draw an active window frame and caption bar, but it does not receive another WM_MDIACTIVATE message.

WM_MDICASCADE 3.0

This message arranges the child windows of a multiple document interface (MDI) client window in a "cascade" format.

Parameter	Description
wParam	Not used.
lParam	Not used.

WM_MDICREATE 3.0

This message causes a multiple document interface (MDI) client window to create a child window.

Parameter	Description
wParam	Not used.
lParam	Contains a long pointer to an **MDICREATESTRUCT** data structure.

Return Value The return value contains the identifier of the new window in the low word and zero in the high word.

Comments The window is created with the style bits WS_CHILD, WS_CLIPSIBLINGS, WS_CLIPCHILDREN, WS_SYSMENU, WS_CAPTION, WS_THICKFRAME, WS_MINIMIZEBOX, and WS_MAXIMIZEBOX, plus additional style bits specified in the **MDICREATESTRUCT** data structure to which *lParam* points. Windows adds the title of the new child window to the window menu of the frame window. An application should create all child windows of the client window with this message.

If a client window receives any message that changes the activation of child windows and the currently active MDI child window is maximized, Windows restores the currently active child and maximizes the newly activated child.

When the MDI child window is created, Windows sends the WM_CREATE message to the window. The *lParam* parameter of the WM_CREATE message contains a pointer to a **CREATESTRUCT** data structure. The **lpCreateParams** field of the **CREATESTRUCT** structure contains a pointer to the **MDICREATESTRUCT** data structure passed with the WM_MDICREATE message that created the MDI child window.

An application should not send a second WM_MDICREATE message while a WM_MDICREATE message is still being processed. For example, it should not send a WM_MDICREATE message while an MDI child window is processing its WM_CREATE message.

WM_MDIDESTROY 3.0

When sent to a multiple document interface (MDI) client window, this message causes a child window to be closed.

Parameter	Description
wParam	Contains the window handle of the child window.
lParam	Not used.

Comments This message removes the title of the child window from the frame window and deactivates the child window. An application should close all MDI child windows with this message.

If a client window receives any message that changes the activation of child windows and the currently active MDI child window is maximized, Windows restores the currently active child and maximizes the newly activated child.

WM_MDIGETACTIVE 3.0

This message returns the current active multiple document interface (MDI) child window, along with a flag indicating whether the child is maximized or not.

Parameter	Description
wParam	Not used.
lParam	Not used.

Return Value The return value contains the handle of the active MDI child window in its low word. If the window is maximized, the high word contains 1; otherwise, the high word is zero.

WM_MDIICONARRANGE 3.0

This message is sent to a multiple document interface (MDI) client window to arrange all minimized document child windows. It does not affect child windows that are not minimized.

Parameter	Description
wParam	Not used.
lParam	Not used.

WM_MDIMAXIMIZE 3.0

This message causes a multiple document interface (MDI) client window to maximize an MDI child window. When a child window is maximized, Windows resizes it to make its client area fill the client window. Windows places the child window's System menu in the frame's menu bar so that the user can restore or close the child window and adds the title of the child window to the frame window title.

Parameter	Description
wParam	Contains the window identifier of the child window.
lParam	Not used.

Comments If an MDI client window receives any message that changes the activation of its child windows, and if the currently active MDI child window is maximized, Windows restores the currently active child and maximizes the newly activated child.

WM_MDINEXT 3.0

This message activates the next multiple document interface (MDI) child window immediately behind the currently active child window and places the currently active window behind all other child windows.

Parameter	Description
wParam	Not used.
lParam	Not used.

Comments If an MDI client window receives any message that changes the activation of its child windows, and if the currently active MDI child window is maximized, Windows restores the currently active child and maximizes the newly activated child.

WM_MDIRESTORE 3.0

This message restores a multiple document interface (MDI) child window from maximized or minimized size.

Parameter	Description
wParam	Contains the window identifier of the child window.
lParam	Not used.

WM_MDISETMENU 3.0

This message replaces the menu of a multiple document interface (MDI) frame window, the Window pop-up menu, or both.

Parameter	Description
wParam	Not used.
lParam	Contains in its low-order word the menu handle (**HMENU**) of the new frame-window menu, and contains in its high-order word the menu handle of the new Window pop-up menu. If either word is zero, the corresponding menu is not changed.

Return Value The return value is the handle of the frame-window menu replaced by this message.

Comments After sending this message, an application must call the **DrawMenuBar** function to update the menu bar.

If this message replaces the Window pop-up menu, MDI child-window menu items are removed from the previous Window menu and added to the new Window pop-up menu.

If an MDI child window is maximized and this message replaces the MDI frame-window menu, the System menu and restore controls are removed from the previous frame-window menu and added to the new menu.

WM_MDITILE 3.0

This message causes a multiple document interface (MDI) client window to arrange all its child windows in a tiled format.

Parameter	Description
wParam	Not used.
lParam	Not used.

WM_MEASUREITEM 3.0

This message is sent to the owner of an owner-draw button, combo box, list box, or menu item when the control is created. When the owner receives the message, the owner fills in the **MEASUREITEM** data structure pointed to by the *lParam* message parameter and returns; this informs Windows of the dimensions of the control. If a list box or combo box is created with the LBS_OWNERDRAWVARIABLE or CBS_OWNERDRAWVARIABLE style, this message is sent to the owner for each item in the control. Otherwise, this message is sent once.

Parameter	Description
wParam	Not used.
lParam	Contains a long pointer to a **MEASUREITEMSTRUCT** data structure that contains the dimensions of the owner-draw control.

Comments Windows sends the WM_MEASUREITEM message to the owner of combo boxes and list boxes created with the OWNERDRAWFIXED style before sending WM_INITDIALOG.

WM_MENUCHAR

This message is sent when the user presses a menu mnemonic character that doesn't match any of the predefined mnemonics in the current menu. It is sent to the window that owns the menu.

Parameter	Description
wParam	Contains the ASCII character that the user pressed.
lParam	The high-order word contains a handle to the selected menu. The low-order word contains the MF_POPUP flag if the menu is a pop-up menu. It contains the MF_SYSMENU flag if the menu is a System menu.

Return Value The high-order word of the return value contains one of the following command codes:

Code	Meaning
0	Informs Windows that it should discard the character that the user pressed, and creates a short beep on the system speaker.
1	Informs Windows that it should close the current menu.
2	Informs Windows that the low-order word of the return value contains the menu item-number for a specific item. This item is selected by Windows.

The low-order word is ignored if the high-order word contains zero or 1. Applications should process this message when accelerators are used to select bitmaps placed in a menu.

WM_MENUSELECT

This message occurs when the user selects a menu item.

Parameter	Description
wParam	Identifies the item selected. If the selected item is a menu item, *wParam* contains the menu-item ID. If the selected item contains a pop-up menu, *wParam* contains the handle of the pop-up menu.
lParam	The low-order word contains a combination of the following menu flags:

Value	Meaning
MF_BITMAP	Item is a bitmap.
MF_CHECKED	Item is checked.
MF_DISABLED	Item is disabled.
MF_GRAYED	Item is grayed.
MF_MOUSESELECT	Item was selected with a mouse.
MF_OWNERDRAW	Item is an owner-draw item.
MF_POPUP	Item contains a pop-up menu.
MF_SYSMENU	Item is contained in the System menu. The high-order word identifies the menu associated with the message.

Comments If the low-order word of the *lParam* parameter contains –1 and the high-order word of the *wParam* parameter contains 0, Windows has closed the menu because the user pressed ESC or clicked outside the menu.

WM_MOUSEACTIVATE

This message occurs when the cursor is in an inactive window and any mouse button is pressed. The parent receives this message only if the child passes it to the **DefWindow-Proc** function.

Parameter	Description
wParam	Contains a handle to the topmost parent window of the window being activated.
lParam	Contains the hit-test area code in the low-order word and the mouse message number in the high-order word. A hit test is a test that determines the location of the cursor.

Comments If the child window passes the message to the **DefWindowProc** function, **DefWindow-Proc** passes this message to a window's parent window before any processing occurs. If the parent window returns TRUE, processing is halted.

For a description of the individual hit-test area codes, see Table 6.2, "Hit-Test Codes."

WM_MOUSEMOVE

This message occurs when the user moves the mouse.

Parameter	Description
wParam	Contains a value that indicates whether various virtual keys are down. It can be any combination of the following values:

Value	Meaning
MK_CONTROL	Set if CONTROL key is down.
MK_LBUTTON	Set if left button is down.
MK_MBUTTON	Set if middle button is down.
MK_RBUTTON	Set if right button is down.
MK_SHIFT	Set if SHIFT key is down.

Parameter	Description
lParam	Contains the *x*- and *y*-coordinates of the cursor. The *x*-coordinate is in the low-order word; the *y*-coordinate is in the high-order word. These coordinates are always relative to the upper-left corner of the window.

Comments The **MAKEPOINT** macro can be used to convert the *lParam* parameter to a **POINT** structure.

WM_MOVE

This message is sent after a window has been moved.

Parameter	Description
wParam	Is not used.
lParam	Contains the new location of the upper-left corner of the client area of the window. This new location is given in screen coordinates for overlapped and pop-up windows and parent-client coordinates for child windows. The *x*-coordinate is in the low-order word; the *y*-coordinate is in the high-order word.

WM_NCACTIVATE

This message is sent to a window when its nonclient area needs to be changed to indicate an active or inactive state.

Parameter	Description
wParam	Specifies when a caption bar or icon needs to be changed to indicate an active or inactive state. The *wParam* parameter is nonzero if an active caption or icon is to be drawn. It is zero for an inactive caption or icon.
lParam	Is not used.

Default Action The **DefWindowProc** function draws a gray caption bar for an inactive window and a black caption bar for an active window.

WM_NCCALCSIZE

This message is sent when the size of a window's client area needs to be calculated.

Parameter	Description
wParam	Is not used.
lParam	Points to a **RECT** data structure that contains the screen coordinates of the window rectangle (including client area, borders, caption, scroll bars, and so on).

Default Action The **DefWindowProc** function calculates the size of the client area, based on the window characteristics (presence of scroll bars, menu, and so on), and places the result in the **RECT** data structure.

WM_NCCREATE

This message is sent prior to the WM_CREATE message when a window is first created.

Parameter	Description
wParam	Contains a handle to the window that is being created.
lParam	Points to the **CREATESTRUCT** data structure for the window.

Return Value The return value is nonzero if the nonclient area is created. It is zero if an error occurs; the **CreateWindow** function will return NULL in this case.

Default Action Scroll bars are initialized (the scroll-bar position and range are set) and the window text is set. Memory used internally to create and maintain the window is allocated.

WM_NCDESTROY

This message informs a window that its nonclient area is being destroyed. The **Destroy-Window** function sends the WM_NCDESTROY message to the window following the WM_DESTROY message. This message is used to free the allocated memory block associated with the window.

Parameter	Description
wParam	Is not used.
lParam	Is not used.

Default Action This message frees any memory internally allocated for the window.

WM_NCHITTEST

This message is sent to the window that contains the cursor (or the window that used the **GetCapture** function to capture the mouse input) every time the mouse is moved.

Parameter	Description
wParam	Is not used.
lParam	Contains the *x*- and *y*-coordinates of the cursor. The *x*-coordinate is in the low-order word; the *y*-coordinate is in the high-order word. These coordinates are always screen coordinates.

Return Value The return value of the **DefWindowProc** function is one of the values given in Table 6.2, indicating the position of the cursor:

Table 6.2 Hit-Test Codes

Code	Meaning
HTBOTTOM	In the lower horizontal border of window.
HTBOTTOMLEFT	In the lower-left corner of window border.
HTBOTTOMRIGHT	In the lower-right corner of window border.
HTCAPTION	In a caption area.
HTCLIENT	In a client area.
HTERROR	Same as HTNOWHERE except that the **DefWindowProc** function produces a system beep to indicate an error.
HTGROWBOX	In a size box.
HTHSCROLL	In the horizontal scroll bar.
HTLEFT	In the left border of window.
HTMENU	In a menu area.
HTNOWHERE	On the screen background or on a dividing line between windows.
HTREDUCE	In a minimize box.
HTRIGHT	In the right border of window.
HTSIZE	Same as HTGROWBOX.
HTSYSMENU	In a control-menu box (close box in child windows).
HTTOP	In the upper horizontal border of window.
HTTOPLEFT	In the upper-left corner of window border.

Table 6.2 Hit-Test Codes *(continued)*

Code	Meaning
HTTOPRIGHT	In the upper-right corner of window border.
HTTRANSPARENT	In a window currently covered by another window.
HTVSCROLL	In the vertical scroll bar.
HTZOOM	In a maximize box.

Comments The **MAKEPOINT** macro can be used to convert the *lParam* parameter to a **POINT** structure.

WM_NCLBUTTONDBLCLK

This message is sent to a window when the user double-clicks the left mouse button while the cursor is within a nonclient area of the window.

Parameter	Description
wParam	Contains the code returned by WM_NCHITTEST (for more information, see the WM_NCHITTEST message, earlier in this chapter).
lParam	Contains a **POINT** data structure that contains the *x*- and *y*-screen coordinates of the cursor position. These coordinates are always relative to the upper-left corner of the screen.

Default Action If appropriate, WM_SYSCOMMAND messages are sent.

WM_NCLBUTTONDOWN

This message is sent to a window when the user presses the left mouse button while the cursor is within a nonclient area of the window.

Parameter	Description
wParam	Contains the code returned by WM_NCHITTEST (for more information, see the WM_NCHITTEST message, earlier in this chapter).

Parameter	Description
lParam	Contains a **POINT** data structure that contains the *x*- and *y*-screen coordinates of the cursor position. These coordinates are always relative to the upper-left corner of the screen.

Default Action If appropriate, WM_SYSCOMMAND messages are sent.

WM_NCLBUTTONUP

This message is sent to a window when the user releases the left mouse button while the cursor is within a nonclient area of the window.

Parameter	Description
wParam	Contains the code returned by WM_NCHITTEST (for more information, see the WM_NCHITTEST message, earlier in this chapter).
lParam	Contains a **POINT** data structure that contains the *x*- and *y*-screen coordinates of the cursor position. These coordinates are always relative to the upper-left corner of the screen.

Default Action If appropriate, WM_SYSCOMMAND messages are sent.

WM_NCMBUTTONDBLCLK

This message is sent to a window when the user double-clicks the middle mouse button while the cursor is within a nonclient area of the window.

Parameter	Description
wParam	Contains the code returned by WM_NCHITTEST (for more information, see the WM_NCHITTEST message, earlier in this chapter).
lParam	Contains a **POINT** data structure that contains the *x*- and *y*-screen coordinates of the cursor position. These coordinates are always relative to the upper-left corner of the screen.

WM_NCMBUTTONDOWN

This message is sent to a window when the user presses the middle mouse button while the cursor is within a nonclient area of the window.

Parameter	Description
wParam	Contains the code returned by WM_NCHITTEST (for more information, see the WM_NCHITTEST message, earlier in this chapter).
lParam	Contains a **POINT** data structure that contains the *x*- and *y*-screen coordinates of the cursor position. These coordinates are always relative to the upper-left corner of the screen.

WM_NCMBUTTONUP

This message is sent to a window when the user releases the left mouse button while the cursor is within a nonclient area of the window.

Parameter	Description
wParam	Contains the code returned by WM_NCHITTEST (for more information, see the WM_NCHITTEST message, earlier in this chapter).
lParam	Contains a **POINT** data structure that contains the *x*- and *y*-screen coordinates of the cursor position. These coordinates are always relative to the upper-left corner of the screen.

WM_NCMOUSEMOVE

This message is sent to a window when the cursor is moved within a nonclient area of the window.

Parameter	Description
wParam	Contains the code returned by WM_NCHITTEST (for more information, see the WM_NCHITTEST message, earlier in this chapter).

Parameter	Description
lParam	Contains a **POINT** data structure that contains the *x*- and *y*-screen coordinates of the cursor position. These coordinates are always relative to the upper-left corner of the screen.

Default Action If appropriate, WM_SYSCOMMAND messages are sent.

WM_NCPAINT

This message is sent to a window when its frame needs painting.

Parameter	Description
wParam	Is not used.
lParam	Is not used.

Default Action The **DefWindowProc** function paints the window frame.

Comments An application can intercept this message and paint its own custom window frame. Remember that the clipping region for a window is always rectangular, even if the shape of the frame is altered.

WM_NCRBUTTONDBLCLK

This message is sent to a window when the user double-clicks the right mouse button while the cursor is within a nonclient area of the window.

Parameter	Description
wParam	Contains the code returned by WM_NCHITTEST (for more information, see the WM_NCHITTEST message, earlier in this chapter).
lParam	Contains a **POINT** data structure that contains the *x*- and *y*-screen coordinates of the cursor position. These coordinates are always relative to the upper-left corner of the screen.

WM_NCRBUTTONDOWN

This message is sent to a window when the user presses the right mouse button while the cursor is within a nonclient area of the window.

Parameter	Description
wParam	Contains the code returned by WM_NCHITTEST (for more information, see the WM_NCHITTEST message, earlier in this chapter).
lParam	Contains a **POINT** data structure that contains the *x*- and *y*-screen coordinates of the cursor position. These coordinates are always relative to the upper-left corner of the screen.

WM_NCRBUTTONUP

This message is sent to a window when the user releases the right mouse button while the cursor is within a nonclient area of the window.

Parameter	Description
wParam	Contains the code returned by WM_NCHITTEST (for more information, see the WM_NCHITTEST message, earlier in this chapter).
lParam	Contains a **POINT** data structure that contains the *x*- and *y*-screen coordinates of the cursor position. These coordinates are always relative to the upper-left corner of the screen.

WM_NEXTDLGCTL

This message is sent to a dialog box's window function, to alter the control focus. The effect of this message is different than that of the **SetFocus** function because WM_NEXTDLGCTL modifies the border around the default button.

Parameter	Description
wParam	If the *lParam* parameter is nonzero, the *wParam* parameter identifies the control that receives the focus. If *lParam* is zero, *wParam* is a flag that indicates whether the next or previous control with tab-stop style receives the focus. If *wParam* is zero, the next control receives the focus; otherwise, the previous control with tab-stop style receives the focus.
lParam	Contains a flag that indicates how Windows uses the *wParam* parameter. If the *lParam* parameter is nonzero, *wParam* is a handle associated with the control that receives the focus; otherwise, *wParam* is a flag that indicates whether the next or previous control with tab-stop style receives the focus.

Comments Do not use the **SendMessage** function to send a WM_NEXTDLGCTL message if your application will concurrently process other messages that set the control focus. Use the **PostMessage** function instead.

WM_PAINT

This message is sent when Windows or an application makes a request to repaint a portion of an application's window. The message is sent either when the **UpdateWindow** function is called or by the **DispatchMessage** function when the application obtains a WM_PAINT message by using the **GetMessage** or **PeekMessage** function.

Parameter	Description
wParam	Is not used.
lParam	Is not used.

WM_PAINTCLIPBOARD

This message is sent when the clipboard contains a data handle for the CF_OWNERDISPLAY format (specifically the clipboard owner should display the clipboard contents) and the Clipboard application's client area needs repainting. The WM_PAINTCLIPBOARD message is sent to the clipboard owner to request repainting of all or part of the Clipboard application's client area.

Parameter	Description
wParam	Contains a handle to the Clipboard-application window.
lParam	The low-order word of the *lParam* parameter identifies a **PAINTSTRUCT** data structure that defines what part of the client area to paint. The high-order word is not used.

Comments

To determine whether the entire client area or just a portion of it needs repainting, the clipboard owner must compare the dimensions of the drawing area given in the **rcpaint** field of the **PAINTSTRUCT** data structure to the dimensions given in the most recent WM_SIZECLIPBOARD message.

An application must use the **GlobalLock** function to lock the memory that contains the **PAINTSTRUCT** data structure. The application should unlock that memory by using the **GlobalUnlock** function before it yields or returns control.

WM_PAINTICON 3.0

This message is sent to a minimized (iconic) window when the icon is to be painted. A window receives this message only if a class icon is defined for the window. Otherwise, WM_PAINT is sent instead. Passing this message to the **DefWindowProc** function permits Windows to paint the icon with the class icon.

Parameter	Description
wParam	Is not used.
lParam	Is not used.

WM_PALETTECHANGED 3.0

This message informs all windows that the window with input focus has realized its logical palette, thereby changing the system palette. This message allows windows without input focus that use a color palette to realize their logical palettes and update their client areas.

Parameter	Description
wParam	Contains the handle of the window that caused the system palette to change.
lParam	Is not used.

Comments

To avoid creating a loop, a window that receives this message should not realize its palette unless it determines that *wParam* does not contain its window handle.

WM_PARENTNOTIFY 3.0

This message is sent to the parent of a child window when the child window is created or destroyed, or when the user has pressed a mouse button while the cursor is over the child window. When the child window is being created, Windows sends WM_PARENTNOTIFY just before the **CreateWindow** or **CreateWindowEx** function that creates the window returns. When the child window is being destroyed, Windows sends the message before any processing to destroy the window takes place.

Parameter	Description
wParam	Specifies the event for which the parent is being notified. It can be any of these values:
lParam	Contains the window handle of the child window in its low-order word and the ID of the child window in its high-order word.

Values for *wParam*:

Value	Meaning
WM_CREATE	The child window is being created.
WM_DESTROY	The child window is being destroyed.
WM_LBUTTONDOWN WM_MBUTTONDOWN WM_RBUTTONDOWN	The user has clicked on a child window.

Comments

This message is also sent to all ancestor windows of the child window, including the top-level window.

This message is sent to the parent of all child windows unless the child has the extended window style WS_EX_NOPARENTNOTIFY; **CreateWindowEx** creates a window with extended window styles. By default, child windows in a dialog box have the WS_EX_NOPARENTNOTIFY style unless the child window was created by calling the **CreateWindowEx** function.

WM_PASTE

This message inserts the data from the clipboard into the control window at the current cursor position. Data are inserted only if the clipboard contains data in CF_TEXT format.

Parameter	Description
wParam	Is not used.
lParam	Is not used.

WM_QUERYDRAGICON 3.0

This message is sent to a minimized (iconic) window which is about to be dragged by the user but which does not have an icon defined for its class.

When the user drags the icon of a window without a class icon, Windows replaces the icon with a default icon cursor. If the application needs a different cursor to be displayed during dragging, it must return the handle of a monochrome cursor compatible with the display driver's resolution. The application can call the **LoadCursor** function to load a cursor from the resources in its executable file and to obtain this handle.

Parameter	Description
wParam	Is not used.
lParam	Is not used.

Return Value The return value contains in its low-order word the handle of the cursor which Windows is to display while the user drags the icon. The return value is NULL if Windows is to display the default icon cursor. The default return value is NULL.

WM_QUERYENDSESSION

This message is sent when the user chooses the End Session command. If any application returns zero, the session is not ended. Windows stops sending WM_QUERYENDSESSION messages as soon as one application returns zero, and sends WM_ENDSESSION messages, with the *wParam* parameter set to zero, to any applications that have already returned nonzero.

Parameter	Description
wParam	Is not used.
lParam	Is not used.

Return Value The return value is nonzero if the application can be conveniently shut down. Otherwise, it is zero.

Default Action The **DefWindowProc** function returns nonzero.

WM_QUERYNEWPALETTE 3.0

This message informs a window that it is about to receive input focus. If the window realizes its logical palette when it receives input focus, the window should return TRUE; otherwise, it should return FALSE.

Parameter	Description
wParam	Is not used.
lParam	Is not used.

Return Value The return value is TRUE if the window realizes its logical palette. Otherwise, it is FALSE.

WM_QUERYOPEN

This message is sent to an icon when the user requests that it be opened into a window.

Parameter	Description
wParam	Is not used.
lParam	Is not used.

Return Value The return value is zero when the application prevents the icon from being opened. It is nonzero when the icon can be opened.

Default Action The **DefWindowProc** function returns nonzero.

W

WM_QUIT

This message indicates a request to terminate an application and is generated when the application calls the **PostQuitMessage** function. It causes the **GetMessage** function to return zero.

Parameter	Description
wParam	Contains the exit code given in the **PostQuitMessage** call.
lParam	Is not used.

WM_RBUTTONDBLCLK

This message occurs when the user double-clicks the right mouse button.

Parameter	Description
wParam	Contains a value that indicates whether various virtual keys are down. It can be any combination of the following values:

Value	Meaning
MK_CONTROL	Set if CONTROL key is down.
MK_LBUTTON	Set if left button is down.
MK_MBUTTON	Set if middle button is down.
MK_RBUTTON	Set if right button is down.
MK_SHIFT	Set if SHIFT key is down.

Parameter	Description
lParam	Contains the *x*- and *y*-coordinates of the cursor. The *x*-coordinate is in the low-order word; the *y*-coordinate is in the high-order word. These coordinates are always relative to the upper-left corner of the window.

Comments Only windows whose window class has CS_DBLCLKS style can receive double-click messages. Windows generates a double-click message when the user presses, releases, and then presses a mouse button again within the system's double-click time limit. Double-clicking actually generates four messages: a down-click message, an up-click message, the double-click message, and another up-click message.

WM_RBUTTONDOWN

This message occurs when the user presses the right mouse button.

Parameter	Description
wParam	Contains a value that indicates whether various virtual keys are down. It can be any combination of the following values:

Value	Meaning
MK_CONTROL	Set if CONTROL key is down.
MK_LBUTTON	Set if left button is down.
MK_MBUTTON	Set if middle button is down.
MK_SHIFT	Set if SHIFT key is down.

Parameter	Description
lParam	Contains the *x*- and *y*-coordinates of the cursor. The *x*-coordinate is in the low-order word; the *y*-coordinate is in the high-order word. These coordinates are always relative to the upper-left corner of the window.

WM_RBUTTONUP

This message occurs when the user releases the right mouse button.

Parameter	Description
wParam	Contains a value that indicates whether various virtual keys are down. It can be any combination of the following values:

Value	Meaning
MK_CONTROL	Set if CONTROL key is down.
MK_LBUTTON	Set if left button is down.
MK_MBUTTON	Set if middle button is down.
MK_SHIFT	Set if SHIFT key is down.

Parameter	Description
lParam	Contains the *x*- and *y*-coordinates of the cursor. The *x*-coordinate is in the low-order word; the *y*-coordinate is in the high-order word. These coordinates are always relative to the upper-left corner of the window.

WM_RENDERALLFORMATS

This message is sent to the application that owns the clipboard when that application is being destroyed.

Parameter	Description
wParam	Is not used.
lParam	Is not used.

Comments The application should render the clipboard data in all the formats it is capable of generating and pass a handle to each format to the **SetClipboardData** function. This ensures that the data in the clipboard can be rendered even though the application has been destroyed.

WM_RENDERFORMAT

This message requests that the clipboard owner format the data last copied to the clipboard in the specified format, and then pass a handle to the formatted data to the clipboard.

Parameter	Description
wParam	Specifies the data format. It can be any one of the formats described with the **SetClipboardData** function.
lParam	Is not used.

WM_SETCURSOR

This message occurs if mouse input is not captured and the mouse causes cursor movement within a window.

Parameter	Description
wParam	Contains a handle to the window that contains the cursor.
lParam	Contains the hit-test area code in the low-order word and the mouse message number in the high-order word.

Comments The **DefWindowProc** function passes the WM_SETCURSOR message to a parent window before processing. If the parent window returns TRUE, further processing is halted. Passing the message to a window's parent window gives the parent window control over the cursor's setting in a child window. The **DefWindowProc** function also uses this message to set the cursor to an arrow if it is not in the client area, or to the registered-class cursor if it is. If the low-order word of the *lParam* parameter is HTERROR and the high-order word of *lParam* is a mouse button-down message, the **MessageBeep** function is called.

The high-order word of *lParam* is zero when the window enters menu mode.

WM_SETFOCUS

This message is sent after a window gains the input focus.

Parameter	Description
wParam	Contains the handle of the window that loses the input focus (may be NULL).
lParam	Is not used.

Comments To display a caret, an application should call the appropriate caret functions at this point.

WM_SETFONT 3.0

This message specifies the font that a dialog box control is to use when drawing text. The best time for the owner of a dialog box control to set the font of the control is when it receives the WM_INITDIALOG message. The application should call the **DeleteObject** function to delete the font when it is no longer needed, such as after the control is destroyed.

The size of the control is not changed as a result of receiving this message. To prevent Windows from clipping text that does not fit within the boundaries of the control, the application should correct the size of the control window before changing the font.

Parameter	Description
wParam	Contains the handle of the font. If this parameter is NULL, the control will draw text using the default system font.
lParam	Specifies whether the control should be redrawn immediately upon setting the font. Setting *lparam* to TRUE causes the control to redraw itself; otherwise, it will not.

Comments

Before Windows creates a dialog box with the DS_SETFONT style, Windows sends the WM_SETFONT message to the dialog-box window before creating the controls. An application creates a dialog box with the DS_SETFONT style by calling any of the following functions:

- **CreateDialogIndirect**
- **CreateDialogIndirectParam**
- **DialogBoxIndirect**
- **DialogBoxIndirectParam**

The **DLGTEMPLATE** data structure which the application passes to these functions must have the DS_SETFONT style set and must contain a **FONTINFO** data structure that defines the font with which the dialog box will draw text.

WM_SETREDRAW

This message is sent by an application to a window in order to allow changes in that window to be redrawn, or to prevent changes in that window from being redrawn.

Parameter	Description
wParam	Specifies the state of the redraw flag. If the *wParam* parameter is nonzero, the redraw flag is set. If *wParam* is zero, the flag is cleared.
lParam	Is not used.

Comments

This message sets or clears the redraw flag. However, it does not direct a list box to update its display. When the redraw flag is set, a control can be redrawn immediately after each change. When the redraw flag is cleared, no redrawing is done. Applications that need to add several names to a list without redrawing until the final name is added should set the redraw flag before adding the final name to the list.

WM_SETTEXT

This message is used to set the text of a window. For edit controls and combo-box edit controls, the text to be set is the content of the edit control. For button controls, the text is the button name. For other windows, the text is the window caption.

Parameter	Description
wParam	Is not used.
lParam	Points to a null-terminated string that is the window text.

Return Value The return value is LB_ERRSPACE (for a list box) or CB_ERRSPACE (for a combo box) if insufficient space is available to set the text in the edit control. It is CB_ERR if this message is sent to a combo box without an edit control.

Comments This message does not change the current selection in the list box of a combo box. An application should use the CB_SELECTSTRING message to select the list-box item which matches the text in the edit control.

WM_SHOWWINDOW

This message is sent when a window is to be hidden or shown. A window is hidden or shown when the **ShowWindow** function is called; when an overlapped window is maximized or restored; or when an overlapped or pop-up window is closed (made iconic) or opened (displayed on the screen). When an overlapped window is closed, all pop-up windows associated with that window are hidden.

Parameter	Description
wParam	Specifies whether a window is being shown. It is nonzero if the window is being shown. It is zero if the window is being hidden.
lParam	Specifies the status of the window being shown. It is zero if the message is sent because of a **ShowWindow** function call. Otherwise, the *lParam* parameter is one of the following values:

Value	Meaning
SW_PARENTCLOSING	Parent window is closing (being made iconic) or a pop-up window is being hidden.
SW_PARENTOPENING	Parent window is opening (being displayed) or a pop-up window is being shown.

Default Action The **DefWindowProc** function hides or shows the window as specified by the message.

WM_SIZE

This message is sent after the size of a window has changed.

Parameter	Description
wParam	Contains a value that defines the type of resizing requested. It can be one of the following values:

Value	Meaning
SIZEFULLSCREEN	Window has been maximized.
SIZEICONIC	Window has been minimized.
SIZENORMAL	Window has been resized, but neither SIZEICONIC nor SIZE-FULLSCREEN applies.
SIZEZOOMHIDE	Message is sent to all pop-up windows when some other window is maximized.
SIZEZOOMSHOW	Message is sent to all pop-up windows when some other window has been restored to its former size.

Parameter	Description
lParam	Contains the new width and height of the client area of the window. The width is in the low-order word; the height is in the high-order word.

Comments If the **SetScrollPos** or **MoveWindow** function is called for a child window as a result of the WM_SIZE message, the *bRedraw* parameter should be nonzero to cause the window to be repainted.

WM_SIZECLIPBOARD

This message is sent when the clipboard contains a data handle for the CF_OWNERDISPLAY format (that is, the clipboard owner should display the clipboard contents) and the clipboard-application window has changed size.

Parameter	Description
wParam	Identifies the clipboard-application window.
lParam	The low-order word of the *lParam* parameter identifies a **RECT** data structure that specifies the area the clipboard owner should paint. The high-order word is not used.

Comments A WM_SIZECLIPBOARD message is sent with a null rectangle (0,0,0,0) as the new size when the clipboard application is about to be destroyed or minimized. This permits the clipboard owner to free its display resources.

An application must use the **GlobalLock** function to lock the memory that contains the **PAINTSTRUCT** data structure. The application should unlock that memory by using the **GlobalUnlock** function before it yields or returns control.

WM_SPOOLERSTATUS 3.0

This message is sent from Print Manager whenever a job is added to or removed from the Print Manager queue.

Parameter	Description
wParam	Is set to SP_JOBSTATUS.
lparam	Specifies in its low-order word the number of jobs remaining in the Print Manager queue. The high-order word is not used.

Comments This message is for informational purposes only.

WM_SYSCHAR

This message results when a WM_SYSKEYUP and WM_SYSKEYDOWN message are translated. It specifies the virtual-key code of the System-menu key.

Parameter	Description
wParam	Contains the ASCII-character key code of a System-menu key.
lParam	Contains the repeat count, scan code, key-transition code, previous key state, and context code, as shown in the following list:

Parameter	Description	
	Bit	**Value**
	0–15 (low-order word)	Repeat count (the number of times the key stroke is repeated as a result of the user holding down the key).
	16–23 (low byte of high-order word)	Scan code (OEM-dependent value).
	24	Extended key, such as a function key or a key on the numeric key pad (1 if it is an extended key, 0 otherwise).
	25–26	Not used.
	27–28	Used internally by Windows.
	29	Context code (1 if the ALT key is held down while the key is pressed, 0 otherwise).
	30	Previous key state (1 if the key is down before the message is sent, 0 if the key is up).
	31	Transition state (1 if the key is being released, 0 if the key is being pressed).

Default Action None.

Comments When the context code is zero, the message can be passed to the **TranslateAccelerator** function, which will handle it as though it were a normal key message instead of a system-key message. This allows accelerator keys to be used with the active window even if the active window does not have the input focus.

For IBM Enhanced 101- and 102-key keyboards, enhanced keys are the right ALT and the right CONTROL keys on the main section of the keyboard; the INSERT, DELETE, HOME, END, PAGE UP, PAGE DOWN and DIRECTION keys in the clusters to the left of the numeric key pad; and the divide (/) and ENTER keys in the numeric key pad. Some other keyboards may support the extended-key bit in the *lParam* parameter.

WM_SYSCOLORCHANGE

This message specifies a change in one or more system colors. Windows sends the message to all top-level windows when a change is made in the system color setting.

Parameter	Description
wParam	Is not used.
lParam	Is not used.

Default Action Windows sends a WM_PAINT message to any window that is affected by a system color change.

Comments Applications that have brushes that use the existing system colors should delete those brushes and re-create them by using the new system colors.

WM_SYSCOMMAND

This message is sent when the user selects a command from the System menu or when the user selects the maximize or minimize box.

Parameter	Description
wParam	Specifies the type of system command requested. It can be any one of the following values:

Value	Meaning
SC_CLOSE	Close the window.
SC_HSCROLL	Scroll horizontally.
SC_KEYMENU	Retrieve a menu through a key stroke.
SC_MAXIMIZE (or SC_ZOOM)	Maximize the window.
SC_MINIMIZE (or SC_ICON)	Minimize the window.
SC_MOUSEMENU	Retrieve a menu through a mouse click.

Parameter	Description	
	Value	**Meaning**
	SC_MOVE	Move the window.
	SC_NEXTWINDOW	Move to the next window.
	SC_PREVWINDOW	Move to the previous window.
	SC_RESTORE	Checkpoint (save the previous coordinates).
	SC_SIZE	Size the window.
	SC_VSCROLL	Scroll vertically.
lParam	Contains the cursor coordinates if a System-menu command is chosen with the mouse. The low-order word contains the *x*-coordinate, and the high-order word contains the *y*-coordinate. Otherwise, this parameter is not used.	

Default Action

The **DefWindowProc** function carries out the System-menu request for the predefined actions specified above.

Comments

In WM_SYSCOMMAND messages, the four low-order bits of the *wParam* parameter are used internally by Windows. When an application tests the value of *wParam*, it must combine the value 0xFFF0 with the *wParam* value by using the bitwise AND operator to obtain the correct result.

The menu items in a System menu can be modified by using the **GetSystemMenu**, **AppendMenu**, **InsertMenu**, and **ModifyMenu** functions. Applications that modify the System menu must process WM_SYSCOMMAND messages. Any WM_SYSCOMMAND messages not handled by the application must be passed to the **DefWindowProc** function. Any command values added by an application must be processed by the application and cannot be passed to **DefWindowProc**.

An application can carry out any system command at any time by passing a WM_SYSCOMMAND message to the **DefWindowProc** function.

Accelerator key strokes that are defined to select items from the System menu are translated into WM_SYSCOMMAND messages; all other accelerator key strokes are translated into WM_COMMAND messages.

WM_SYSDEADCHAR

This message results when a WM_SYSKEYUP and WM_SYSKEYDOWN message are translated. It specifies the character value of a dead key.

Parameter	Description
wParam	Contains the dead-key character value.
lParam	Contains a repeat count and an auto-repeat count. The low-order word contains the repeat count; the high-order word contains the auto-repeat count.

WM_SYSKEYDOWN

This message is sent when the user holds down the ALT key and then presses another key. It also occurs when no window currently has the input focus; in this case, the WM_SYSKEYDOWN message is sent to the active window. The window that receives the message can distinguish between these two contexts by checking the context code in the *lParam* parameter.

Parameter	Description
wParam	Contains the virtual-key code of the key being pressed.
lParam	Contains the repeat count, scan code, key-transition code, previous key state, and context code, as shown in the following list:

Bit	Value
0–15 (low-order word)	Repeat count (the number of times the key stroke is repeated as a result of the user holding down the key).
16–23 (low byte of high-order word)	Scan code (OEM-dependent value).
24	Extended key, such as a function key or a key on the numeric key pad (1 if it is an extended key).
25–26	Not used.
27–28	Used internally by Windows.

Parameter	Description	
	Bit	**Value**
	29	Context code (1 if the ALT key is held down while the key is pressed, 0 otherwise).
	30	Previous key state (1 if the key is down before the message is sent, 0 if the key is up).
	31	Transition state (1 if the key is being released, 0 if the key is being pressed).

For WM_SYSKEYDOWN messages, the key-transition bit (bit 31) is 0. The context-code bit (bit 29) is 1 if the ALT key is down while the key is pressed; it is 0 if the message is sent to the active window because no window has the input focus.

Comments

When the context code is zero, the message can be passed to the **TranslateAccelerator** function, which will handle it as though it were a normal key message instead of a system-key message. This allows accelerator keys to be used with the active window even if the active window does not have the input focus.

Because of auto-repeat, more than one WM_SYSKEYDOWN message may occur before a WM_SYSKEYUP message is sent. The previous key state (bit 30) can be used to determine whether the WM_SYSKEYDOWN message indicates the first down transition or a repeated down transition.

For IBM Enhanced 101- and 102-key keyboards, enhanced keys are the right ALT and the right CONTROL keys on the main section of the keyboard; the INSERT, DELETE, HOME, END, PAGE UP, PAGE DOWN and DIRECTION keys in the clusters to the left of the numeric key pad; and the divide (/) and ENTER keys in the numeric key pad. Some other keyboards may support the extended-key bit in the *lParam* parameter.

WM_SYSKEYUP

This message is sent when the user releases a key that was pressed while the ALT key was held down. It also occurs when no window currently has the input focus; in this case, the WM_SYSKEYUP message is sent to the active window. The window that receives the message can distinguish between these two contexts by checking the context code in the *lParam* parameter.

Parameter	Description
wParam	Contains the virtual-key code of the key being released.
lParam	Contains the repeat count, scan code, key-transition code, previous key state, and context code, as shown in the following list:

Bit	Value
0–15 (low-order word)	Repeat count (the number of times the key stroke is repeated as a result of the user holding down the key).
16–23 (low byte of high-order word)	Scan code (OEM-dependent value).
24	Extended key, such as a function key or a key on the numeric key pad (1 if it is an extended key).
25–26	Not used.
27–28	Used internally by Windows.
29	Context code (1 if the ALT key is held down while the key is pressed, 0 otherwise).
30	Previous key state (1 if the key is down before the message is sent, 0 if the key is up).
31	Transition state (1 if the key is being released, 0 if the key is being pressed).

For WM_SYSKEYUP messages, the key-transition bit (bit 31) is 1. The context-code bit (bit 29) is 1 if the ALT key is down while the key is pressed; it is 0 if the message is sent to the active window because no window has the input focus.

Comments

When the context code is zero, the message can be passed to the **TranslateAccelerator** function, which will handle it as though it were a normal key message instead of a system-key message. This allows accelerator keys to be used with the active window even if the active window does not have the input focus.

For IBM Enhanced 101- and 102-key keyboards, enhanced keys are the right ALT and the right CONTROL keys on the main section of the keyboard; the INSERT, DELETE, HOME, END, PAGE UP, PAGE DOWN and DIRECTION keys in the clusters to the left of the numeric key

pad; and the divide (/) and ENTER keys in the numeric key pad. Some other keyboards may support the extended-key bit in the *lParam* parameter.

For non-USA Enhanced 102-key keyboards, the right ALT key is handled as a CONTROL-ALT key. The following shows the sequence of messages which result when the user presses and releases this key:

Order	Message	Virtual-key code (*lParam*)
1	WM_KEYDOWN	VK_CONTROL
2	WM_KEYDOWN	VK_MENU
3	WM_KEYUP	VK_CONTROL
4	WM_SYSKEYUP	VK_MENU

WM_TIMECHANGE

This message occurs when an application makes a change (or set of changes) to the system time. Any application that changes the system time should send this message to all top-level windows.

Parameter	Description
wParam	Is not used.
lParam	Is not used.

Comments To send the WM_TIMECHANGE message to all top-level windows, an application can use the **SendMessage** function with the *hWnd* parameter set to 0xFFFF.

WM_TIMER

This message occurs when the time limit set for a given timer has elapsed.

Parameter	Description
wParam	Contains the timer ID, an integer value that identifies the timer.
lParam	Points to a function that was passed to the **SetTimer** function when the timer was created. If the *lParam* parameter is not NULL, Windows calls the specified function directly, instead of sending the WM_TIMER message to the window function.

W

WM_UNDO

This message undoes the last operation. When sent to an edit control, the previously deleted text is restored or the previously added text is deleted.

Parameter	Description
wParam	Is not used.
lParam	Is not used.

WM_VKEYTOITEM

This message is sent by a list box with the LBS_WANTKEYBOARDINPUT style to its owner in response to a WM_KEYDOWN message.

Parameter	Description
wParam	Contains the virtual-key code of the key which the user pressed.
lParam	Contains the current caret position in its high-order word and the window handle of the list box in its low-order word.

Return Value The return value specifies the action which the application performed in response to the message. A return value of –2 indicates that the application handled all aspects of selecting the item and wants no further action by the list box. A return value of –1 indicates that the list box should perform the default action in response to the key stroke. A return value of zero or greater specifies the index of an item in the list box and indicates that the list box should perform the default action for the key stroke on the given item.

WM_VSCROLL

This message is sent when the user clicks the vertical scroll bar.

Parameter	Description
wParam	Contains a scroll-bar code that specifies the user's scrolling request. It can be any one of the following values:

Value	Meaning
SB_BOTTOM	Scroll to bottom.
SB_ENDSCROLL	End scroll.
SB_LINEDOWN	Scroll one line down.
SB_LINEUP	Scroll one line up.
SB_PAGEDOWN	Scroll one page down.
SB_PAGEUP	Scroll one page up.
SB_THUMBPOSITION	Scroll to absolute position. The current position is provided in the low-order word of *lParam*.
SB_THUMBTRACK	Drag thumb to specified position. The current position is provided in the low-order word of *lParam*.
SB_TOP	Scroll to top.

Parameter	Description
lParam	If the message is sent by a scroll-bar control, the high-order word of the *lParam* parameter identifies the control. If the message is sent as a result of the user clicking a pop-up window's scroll bar, the high-order word is not used.

Comments

The SB_THUMBTRACK message typically is used by applications that give some feedback while the thumb is being dragged.

If an application scrolls the document in the window, it must also reset the position of the thumb by using the **SetScrollPos** function.

WM_VSCROLLCLIPBOARD

This message is sent when the clipboard contains a data handle for the CF_OWNERDISPLAY format (that is, the clipboard owner should display the clipboard contents) and an event occurs in the clipboard-application's vertical scroll bar.

Parameter	Description
wParam	Contains a handle to the clipboard-application window.
lParam	Contains one of the following scroll-bar codes in the low-order word:

Value	Meaning
SB_BOTTOM	Scroll to lower right.
SB_ENDSCROLL	End scroll.
SB_LINEDOWN	Scroll one line down.
SB_LINEUP	Scroll one line up.
SB_PAGEDOWN	Scroll one page down.
SB_PAGEUP	Scroll one page up.
SB_THUMBPOSITION	Scroll to absolute position.
SB_TOP	Scroll to upper left.

The high-order word of the *lParam* parameter contains the thumb position if the scroll-bar code is SB_THUMBPOSITION. Otherwise, the high-order word is not used.

Comments The clipboard owner should use the **InvalidateRect** function or repaint as desired. The scroll bar position should also be reset.

WM_WININICHANGE

This message is sent when the Windows initialization file, WIN.INI, changes. Any application that makes a change to WIN.INI should send this message to all top-level windows.

Parameter	Description
wParam	Is not used.
lParam	Points to a string that specifies the name of the section that has changed (the string does not include the square brackets).

Comments

To send the WM_WININICHANGE message to all top-level windows, an application can use the **SendMessage** function with the *hWnd* parameter set to 0xFFFF.

Although it is incorrect to do so, some applications send this message with *lParam* set to NULL. If an application receives this message with a NULL *lParam*, it should check all sections in WIN.INI that affect the application.

Volume 2

Part 3 General Reference

Part 3 provides general reference information on components of the Windows application programming interface that are in addition to the functions and messages described in the preceding parts.

CHAPTERS

Chapter 7 Data Types and Structures

This chapter describes the data types and structures used by Microsoft Windows functions and messages. It contains two parts: a table of data types and a list of Windows data structures, each arranged alphabetically.

7.1 Data Types

The data types in the following list are key words that define the size and meaning of parameters and return values associated with Windows functions. This list contains character, integer, and Boolean types, pointer types, and handles. The character, integer, and Boolean types are common to most C compilers. Most of the pointer-type names begin with either a P prefix (for short pointers) or an LP prefix (for long pointers). A short pointer accesses data within the current data segment; a long pointer contains a 32-bit segment/offset value. A Windows application uses a handle to refer to a resource that has been loaded into memory. Windows provides access to these resources through internally maintained tables that contain individual entries for each handle. Each entry in the handle table contains the address of the resource and a means of identifying the resource type. The Windows data types are defined in the following list:

Type	Definition
BOOL	16-bit Boolean value.
BYTE	Unsigned 8-bit integer.
char	ASCII character or a signed 8-bit integer.
DWORD	Unsigned 32-bit integer or a segment/offset address.
FAR	Data-type attribute that can be used to create a long pointer.
FARPROC	Long pointer to a function obtained by calling the **MakeProcInstance** function.

Type	Definition
GLOBALHANDLE	Handle to global memory. It is a 16-bit index to a block of memory allocated from the system's global heap.
HANDLE	General handle. It represents a 16-bit index to a table entry that identifies program data.
HBITMAP	Handle to a physical bitmap. It is a 16-bit index to GDI's physical drawing objects.
HBRUSH	Handle to a physical brush. It is a 16-bit index to GDI's physical drawing objects.
HCURSOR	Handle to a cursor resource. It is a 16-bit index to a resource-table entry.
HDC	Handle to a display context. It is a 16-bit index to GDI's device-context tables.
HFONT	Handle to a physical font. It is a 16-bit index to GDI's physical drawing objects.
HICON	Handle to an icon resource. It is a 16-bit index to a resource-table entry.
HMENU	Handle to a menu resource. It is a 16-bit index to a resource-table entry.
HPALETTE	Handle to a logical palette. It is a 16-bit index to GDI's physical drawing objects.
HPEN	Handle to a physical pen. It is a 16-bit index to GDI's physical drawing objects.
HRGN	Handle to a physical region. It is a 16-bit index to GDI's physical drawing objects.
HSTR	Handle to a string resource. It is a 16-bit index to a resource-table entry.
int	Signed 16-bit integer.

Type	Definition
LOCALHANDLE	Handle to local memory. It is a 16-bit index to a block of memory allocated from the application's local heap.
long	Signed 32-bit integer.
LONG	Signed 32-bit integer.
LPBITMAP	Long pointer to a **BITMAP** data structure.
LPBITMAPCOREHEADER	Long pointer to a **BITMAPCOREHEADER** data structure.
LPBITMAPCOREINFO	Long pointer to a **BITMAPCOREINFO** data structure.
LPBITMAPFILEHEADER	Long pointer to a **BITMAPFILEHEADER** data structure.
LPBITMAPINFO	Long pointer to a **BITMAPINFO** data structure.
LPBITMAPINFOHEADER	Long pointer to a **BITMAPINFOHEADER** data structure.
LPCOMPAREITEMSTRUCT	Long pointer to a **COMPAREITEMSTRUCT** data structure.
LPCREATESTRUCT	Long pointer to a **CREATESTRUCT** data structure.
LPDELETEITEMSTRUCT	Long pointer to a **DELETEITEMSTRUCT** data structure.
LPDRAWITEMSTRUCT	Long pointer to a **DRAWITEMSTRUCT** data structure.
LPHANDLETABLE	Long pointer to a **HANDLETABLE** data structure.
LPINT	Long pointer to a signed 16-bit integer.
LPLOGBRUSH	Long pointer to a **LOGBRUSH** data structure.
LPLOGFONT	Long pointer to a **LOGFONT** data structure.
LPLOGPALETTE	Long pointer to a **LOGPALETTE** data structure.

Type	Definition
LPLOGPEN	Long pointer to a **LOGPEN** data structure.
LPMEASUREITEMSTRUCT	Long pointer to a **MEASUREITEMSTRUCT** data structure.
LPMETAFILEPICT	Long pointer to a **METAFILEPICT** data structure.
LPMSG	Long pointer to a **MSG** data structure.
LPOFSTRUCT	Long pointer to an **OFSTRUCT** data structure.
LPPAINTSTRUCT	Long pointer to a **PAINTSTRUCT** data structure.
LPPALETTEENTRY	Long pointer to a **PALETTEENTRY** data structure.
LPPOINT	Long pointer to a **POINT** data structure.
LPRECT	Long pointer to a **RECT** data structure.
LPRESOURCELIST	Long pointer to one or more **RESOURCESTRUCT** data structures.
LPSTR	Long pointer to a character string.
LPTEXTMETRIC	Long pointer to a **TEXTMETRIC** data structure.
LPVOID	Long pointer to an undefined data type.
LPWNDCLASS	Long pointer to a **WNDCLASS** data structure.
NEAR	Data-type attribute that can be used to create a short pointer.
NPSTR	Near pointer to a character string.
PINT	Pointer to a signed 16-bit integer.
PSTR	Pointer to a character string.
PWORD	Pointer to an unsigned 16-bit integer.
short	Signed 16-bit integer.

Type	Definition
void	Empty value. It is used with a function to specify no return value.
WORD	Unsigned 16-bit integer.

7.2 Data Structures

This section lists data structures that are used by Windows. The data structures are presented in alphabetical order. The structure definition is given, followed by a description of each field.

BITMAP

Bitmap Data Structure

The **BITMAP** structure defines the height, width, color format, and bit values of a logical bitmap.

```
typedef struct tagBITMAP {
   short      bmType;
   short      bmWidth;
   short      bmHeight;
   short      bmWidthBytes;
   BYTE       bmPlanes;
   BYTE       bmBitsPixel;
   LPSTR      bmBits;
} BITMAP;
```

The **BITMAP** structure has the following fields:

Field	Description
bmType	Specifies the bitmap type. For logical bitmaps, the **bmType** field must be zero.
bmWidth	Specifies the width of the bitmap (in pixels). The width must be greater than zero.
bmHeight	Specifies the height of the bitmap (in raster lines). The height must be greater than zero.
bmWidthBytes	Specifies the number of bytes in each raster line. This value must be an even number since the graphics device interface (GDI) assumes that the bit values of a bitmap form an array of integer (two-byte) values. In other words, **bmWidthBytes** × 8 must be the next multiple of 16 greater than or equal to the **bmWidth** field.
bmPlanes	Points to the number of color planes in the bitmap.
bmBitsPixel	Points to the number of adjacent color bits on each plane needed to define a pixel.
bmBits	Points to the location of the bit values for the bitmap. The **bmBits** field must be a long pointer to an array of character (one-byte) values.

Comments The currently used bitmap formats are monochrome and color. The monochrome bitmap uses a one-bit, one-plane format. Each scan is a multiple of 16 bits.

Scans are organized as follows for a monochrome bitmap of height *n*:

```
Scan 0
Scan 1
.
.
.
Scan n-2
Scan n-1
```

The pixels on a monochrome device are either black or white. If the corresponding bit in the bitmap is 1, the pixel is turned on (white); if the corresponding bit in the bitmap is zero, the pixel is turned off (black).

All devices that have the RC_BITBLT bit set in the device capabilities support bitmaps.

Each device has its own unique color format. In order to transfer a bitmap from one device to another, use **GetDIBits** and **SetDIBits**.

See Also The **CreateBitmapIndirect** and **GetObject** functions in Chapter 4, "Functions Directory," in *Reference, Volume 1.*

BITMAPCOREHEADER 3.0

Device-Independent Bitmap Format Information

The **BITMAPCOREHEADER** structure contains information about the dimensions and color format of a device-independent bitmap that is compatible with Microsoft OS/2 Presentation Manager versions 1.1 and 1.2 bitmaps.

```
typedef struct tagBITMAPCOREHEADER {
        DWORD   bcSize;
        WORD    bcWidth;
        WORD    bcHeight;
        WORD    bcPlanes;
        WORD    bcBitCount;
} BITMAPCOREHEADER;
```

The **BITMAPCOREHEADER** structure has the following fields:

Field	Description
bcSize	Specifies the number of bytes required by the **BITMAPCOREHEADER** structure.
bcWidth	Specifies the width of the bitmap in pixels.
bcHeight	Specifies the height of the bitmap in pixels.

Field	Description
bcPlanes	Specifies the number of planes for the target device and must be set to 1.
bcBitCount	Specifies the number of bits per pixel. This value must be 1, 4, 8, or 24.

Comments

The **BITMAPCOREINFO** data structure combines the **BITMAPCOREHEADER** structure and a color table to provide a complete definition of the dimensions and colors of a device-independent bitmap. See the description of the **BITMAPCOREINFO** data structure for more information about specifying a device-independent bitmap.

An application should use the information stored in the **bcSize** field to locate the color table in a **BITMAPCOREINFO** data structure with a method such as the following:

```
pColor = ((LPSTR) pBitmapCoreInfo + (WORD) (pBitmapCoreInfo -> bcSize))
```

BITMAPCOREINFO 3.0

Device-Indpendent Bitmap Information

The **BITMAPCOREINFO** structure fully defines the dimensions and color information for a device-independent bitmap that is compatible with Microsoft OS/2 Presentation Manager versions 1.1 and 1.2 bitmaps.

```
typedef struct _BITMAPCOREINFO {
        BITMAPCOREHEADER  bmciHeader;
        RGBTRIPLE         bmciColors[];
} BITMAPCOREINFO;
```

The **BITMAPCOREINFO** structure contains the following fields:

Field	Description
bmciHeader	Specifies a **BITMAPCOREHEADER** data structure that contains information about the dimensions and color format of a device-independent bitmap.
bmciColors	Specifies an array of **RGBTRIPLE** data structures that define the colors in the bitmap.

Comments

An OS/2 Presentation Manager device-independent bitmap consists of two distinct parts: a **BITMAPCOREINFO** data structure that describes the dimensions and colors of the bitmap, and an array of bytes which define the pixels of the bitmap. The bits in the array are packed together, but each scan line must be zero-padded to end on a **LONG** boundary. Segment boundaries can appear anywhere in the bitmap, however. The origin of the bitmap is the lower-left corner.

The **bcBitCount** field of the **BITMAPCOREHEADER** structure determines the number of bits which define each pixel and the maximum number of colors in the bitmap. This field may be set to any of the following values:

Value	Meaning
1	The bitmap is monochrome, and the **bmciColors** field must contain two entries. Each bit in the bitmap array represents a pixel. If the bit is clear, the pixel is displayed with the color of the first entry in the **bmciColors** table; if the bit is set, the pixel has the color of the second entry in the table.
4	The bitmap has a maximum of 16 colors, and the **bmciColors** field contains 16 entries. Each pixel in the bitmap is represented by a four-bit index into the color table. For example, if the first byte in the bitmap is 0x1F, then the byte represents two pixels. The first pixel contains the color in the second table entry, and the second pixel contains the color in the 16th table entry.
8	The bitmap has a maximum of 256 colors, and the **bmciColors** field contains 256 entries. In this case, each byte in the array represents a single pixel.
24	The bitmap has a maximum of 2^{24} colors. The **bmciColors** field is NULL, and each three bytes in the bitmap array represents the relative intensities of red, green, and blue, respectively, of a pixel.

The colors in the **bmciColors** table should appear in order of importance.

Alternatively, for functions that use device-independent bitmaps, the **bmciColors** field can be an array of 16-bit unsigned integers that specify an index into the currently realized logical palette instead of explicit RGB values. In this case, an application using the bitmap must call device-independent bitmap functions with the *wUsage* parameter set to DIB_PAL_COLORS.

Note The **bmciColors** field should not contain palette indexes if the bitmap is to be stored in a file or transferred to another application. Unless the application uses the bitmap exclusively and under its complete control, the bitmap color table should contain explicit RGB values.

BITMAPFILEHEADER 3.0

Bitmap File Information

The **BITMAPFILEHEADER** data structure contains information about the type, size, and layout of a device-independent bitmap (DIB) file.

```
typedef struct tagBITMAPFILEHEADER {
        WORD    bfType;
        DWORD   bfSize;
        WORD    bfReserved1;
        WORD    bfReserved2;
        DWORD   bfOffBits;
} BITMAPFILEHEADER;
```

The **BITMAPFILEHEADER** data structure contains the following fields:

Field	Description
bfType	Specifies the type of file. It must be BM.
bfSize	Specifies the size in **DWORD**s of the file.
bfReserved1	Is reserved and must be set to zero.
bfReserved2	Is reserved and must be set to zero.
bfOffBits	Specifies in bytes the offset from the **BITMAPFILEHEADER** of the actual bitmap in the file.

Comments A **BITMAPINFO** or **BITMAPCOREINFO** data structure immediately follows the **BITMAPFILEHEADER** structure in the DIB file.

BITMAPINFO 3.0

Device-Indpendent Bitmap Information

The **BITMAPINFO** structure fully defines the dimensions and color information for a Windows 3.0 device-independent bitmap.

```
typedef struct tagBITMAPINFO {
   BITMAPINFOHEADER    bmiHeader;
   RGBQUAD             bmiColors[1];
} BITMAPINFO;
```

The **BITMAPINFO** structure contains the following fields:

Field	Description
bmiHeader	Specifies a **BITMAPINFOHEADER** data structure that contains information about the dimensions and color format of a device-independent bitmap.
bmiColors	Specifies an array of **RGBQUAD** data structures that define the colors in the bitmap.

Comments

A Windows 3.0 device-independent bitmap consists of two distinct parts: a **BITMAPINFO** data structure that describes the dimensions and colors of the bitmap, and an array of bytes that define the pixels of the bitmap. The bits in the array are packed together, but each scan line must be zero-padded to end on a **LONG** boundary. Segment boundaries can appear anywhere in the bitmap, however. The origin of the bitmap is the lower-left corner.

The **biBitCount** field of the **BITMAPINFOHEADER** structure determines the number of bits which define each pixel and the maximum number of colors in the bitmap. This field may be set to any of the following values:

Value	Meaning
1	The bitmap is monochrome, and the **bmiColors** field must contain two entries. Each bit in the bitmap array represents a pixel. If the bit is clear, the pixel is displayed with the color of the first entry in the **bmiColors** table; if the bit is set, the pixel has the color of the second entry in the table.
4	The bitmap has a maximum of 16 colors, and the **bmiColors** field contains up to 16 entries. Each pixel in the bitmap is represented by a four-bit index into the color table. For example, if the first byte in the bitmap is 0x1F, then the byte represents two pixels. The first pixel contains the color in the second table entry, and the second pixel contains the color in the 16th table entry.
8	The bitmap has a maximum of 256 colors, and the **bmiColors** field contains up to 256 entries. In this case, each byte in the array represents a single pixel.

Value	Meaning
24	The bitmap has a maximum of 2^{24} colors. The **bmiColors** field is NULL, and each three bytes in the bitmap array represents the relative intensities of red, green, and blue, respectively, of a pixel.

The **biClrUsed** field of the **BITMAPINFOHEADER** structure specifies the number of color indexes in the color table actually used by the bitmap. If the **biClrUsed** field is set to 0, the bitmap uses the maximum number of colors corresponding to the value of the **biBitCount** field.

The colors in the **bmiColors** table should appear in order of importance.

Alternatively, for functions that use device-independent bitmaps, the **bmiColors** field can be an array of 16-bit unsigned integers that specify an index into the currently realized logical palette instead of explicit RGB values. In this case, an application using the bitmap must call device-independent bitmap functions with the *wUsage* parameter set to DIB_PAL_COLORS.

Note The **bmiColors** field should not contain palette indices if the bitmap is to be stored in a file or transferred to another application. Unless the application uses the bitmap exclusively and under its complete control, the bitmap color table should contain explicit RGB values.

BITMAPINFOHEADER 3.0

Device-Independent Bitmap Format Information

The **BITMAPINFOHEADER** structure contains information about the dimensions and color format of a Windows 3.0 device-independent bitmap.

```
typedef struct tagBITMAPINFOHEADER{
   DWORD  biSize;
   DWORD  biWidth;
   DWORD  biHeight;
   WORD   biPlanes;
   WORD   biBitCount
   DWORD  biCompression;
   DWORD  biSizeImage;
   DWORD  biXPelsPerMeter;
   DWORD  biYPelsPerMeter;
   DWORD  biClrUsed;
   DWORD  biClrImportant;
} BITMAPINFOHEADER;
```

The **BITMAPINFOHEADER** structure has the following fields:

Field	Description
biSize	Specifies the number of bytes required by the **BITMAP-INFOHEADER** structure.
biWidth	Specifies the width of the bitmap in pixels.
biHeight	Specifies the height of the bitmap in pixels.
biPlanes	Specifies the number of planes for the target device and must be set to 1.
biBitCount	Specifies the number of bits per pixel. This value must be 1, 4, 8, or 24.
biCompression	Specifies the type of compression for a compressed bitmap. It can be one of the following values:

Value	Meaning
BI_RGB	Specifies that the bitmap is not compressed.
BI_RLE8	Specifies a run-length encoded format for bitmaps with 8 bits per pixel. The compression format is a two-byte format consisting of a count byte followed by a byte containing a color index. See the following "Comments" section for more information.
BI_RLE4	Specifies a run-length encoded format for bitmaps with 4 bits per pixel. The compression format is a two-byte format consisting of a count byte followed by two word-length color indexes. See the following "Comments" section for more information.

Field	Description
biSizeImage	Specifies the size in bytes of the image.
biXPelsPerMeter	Specifies the horizontal resolution in pixels per meter of the target device for the bitmap. An application can use this value to select a bitmap from a resource group that best matches the characteristics of the current device.
biYPelsPerMeter	Specifies the vertical resolution in pixels per meter of the target device for the bitmap.

Field	Description
biClrUsed	Specifies the number of color indexes in the color table actually used by the bitmap. If this value is 0, the bitmap uses the maximum number of colors corresponding to the value of the **biBitCount** field. See the description of the **BITMAPINFO** data structure earlier in this chapter for more information on the maximum sizes of the color table. If **biClrUsed** is nonzero, then the **biClrUsed** field specifies the actual number of colors which the graphics engine or device driver will access if the **biBitCount** field is less than 24. If the **biBitCount** field is set to 24, the **biClrUsed** field specifies the size of the reference color table used to optimize performance of Windows color palettes. If the bitmap is a "packed" bitmap (that is, a bitmap in which the bitmap array immediately follows the **BITMAPINFO** header and which is referenced by a single pointer), the **biClrUsed** field must be set to 0 or to the actual size of the color table.
biClrImportant	Specifies the number of color indexes that are considered important for displaying the bitmap. If this value is 0, then all colors are important.

Comments

The **BITMAPINFO** data structure combines the **BITMAPINFOHEADER** structure and a color table to provide a complete definition of the dimensions and colors of a Windows 3.0 device-independent bitmap. See the description of the **BITMAPINFO** data structure for more information about specifying a Windows 3.0 device-independent bitmap.

An application should use the information stored in the **biSize** field to locate the color table in a **BITMAPINFO** data structure with a method such as the following:

```
pColor = ((LPSTR) pBitmapInfo + (WORD) (pBitmapInfo -> biSize))
```

Bitmap Compression Formats

Windows supports formats for compressing bitmaps that define their colors with 8 bits per pixel and with 4 bits per pixel. Compression reduces the disk and memory storage required for the bitmap. The following paragraphs describe these formats.

When the **biCompression** field is set to BI_RLE8, the bitmap is compressed using a run-length encoding format for an 8-bit bitmap. This format may be compressed in either of two modes:

- Encoded
- Absolute

Both modes can occur anywhere throughout a single bitmap.

Encoded mode consists of two bytes: the first byte specifies the number of consecutive pixels to be drawn using the color index contained in the second byte. In addition, the first byte of the pair can be set to zero to indicate an escape that denotes an end of line, end of bitmap, or a delta. The interpretation of the escape depends on the value of the second byte of the pair. The following list shows the meaning of the second byte:

Second Byte Of Escape	Meaning
0	End of line.
1	End of bitmap.
2	Delta. The two bytes following the escape contain unsigned values indicating the horizontal and vertical offset of the next pixel from the current position.

Absolute mode is signalled by the first byte set to zero and the second byte set to a value between 03H and FFH. In absolute mode, the second byte represents the number of bytes which follow, each of which contains the color index of a single pixel. When the second byte is set to 2 or less, the escape has the same meaning as in encoded mode. In absolute mode, each run must be aligned on a word boundary.

The following example shows the hexadecimal values of an 8-bit compressed bitmap:

```
03 04 05 06 00 03 45 56 67 00 02 78 00 02 05 01
02 78 00 00 09 1E 00 01
```

This bitmap would expand as follows (two-digit values represent a color index for a single pixel):

```
04 04 04
06 06 06 06 06
45 56 67
78 78
move current position 5 right and 1 down
78 78
end of line
1E 1E 1E 1E 1E 1E 1E 1E 1E
end of RLE bitmap
```

When the **biCompression** field is set to BI_RLE4, the bitmap is compressed using a run-length encoding format for a 4-bit bitmap, which also uses encoded and absolute modes. In encoded mode, the first byte of the pair contains the number of pixels to be drawn using the color indexes in the second byte. The second byte contains two color indexes, one in its high-order nibble (that is, its low-order four bits) and one in its low-order nibble. The first of the pixels is drawn using the color specified by the high-order nibble, the second is drawn using the color in the low-order nibble, the third is drawn with the color in the high-order nibble, and so on, until all the pixels specified by the first byte have been drawn.

In absolute mode, the first byte contains zero, the second byte contains the number of color indexes that follow, and subsequent bytes contain color indexes in their high- and low-order nibbles, one color index for each pixel. In absolute mode, each run must be aligned on a word boundary. The end-of-line, end-of-bitmap, and delta escapes also apply to BI_RLE4.

The following example shows the hexadecimal values of a 4-bit compressed bitmap:

```
03 04 05 06 00 06 45 56 67 00 04 78 00 02 05 01
04 78 00 00 09 1E 00 01
```

This bitmap would expand as follows (single-digit values represent a color index for a single pixel):

```
0 4 0
0 6 0 6 0
4 5 5 6 6 7
7 8 7 8
move current position 5 right and 1 down
7 8 7 8
end of line
1 E 1 E 1 E 1 E 1
end of RLE bitmap
```

CLIENTCREATESTRUCT 3.0

MDI Client Window Creation Structure

The **CLIENTCREATESTRUCT** data structure contains information about the menu and first multiple document interface (MDI) child window of an MDI client window. An application passes a long pointer to this structure as the *lpParam* parameter of the **CreateWindow** function when creating an MDI client window.

```
typedef struct tagCLIENTCREATESTRUCT
        {
          HMENU   hWindowMenu;
          WORD    idFirstChild;
        } CLIENTCREATESTRUCT;
```

The **CLIENTCREATESTRUCT** structure contains the following fields:

Field	Description
hWindowMenu	Is the menu handle of the application's Window menu. An application can retrieve this handle from the MDI frame window's menu using the **GetSubMenu** function.
idFirstChild	Is the child window ID of the first MDI child window created. Windows increments the ID for each additional MDI child window that the application creates, and reassigns identifiers when the application destroys a window to keep the range of identifiers continuous. These identifiers are used in WM_COMMAND messages to the application's MDI frame window when a child window is selected from the Window menu, and should not conflict with any other command identifiers.

COLORREF

Color Specification

A **COLORREF** color value is a long integer that specifies a color. GDI functions that require a color (such as **CreatePen** and **FloodFill**) accept a **COLORREF** value as a parameter. Depending on how an application uses the **COLORREF** value, the value has three distinct forms. It may specify any of the following:

- Explicit values for red, green, and blue (RGB)
- An index into a logical color palette
- A palette-relative RGB value

Explict RGB

When specifying an explicit RGB value, the **COLORREF** value has the following hexadecimal form:

```
0x00bbggrr
```

The low-order byte contains a value for the relative intensity of red; the second byte contains a value for green, and the third byte contains a value for blue. The high-order byte must be zero. The maximum value for a single byte is FF (hexadecimal). The following list illustrates the hexadecimal values that produce the indicated colors.

Value	Color
0x000000FF	Pure red
0x0000FF00	Pure green

Value	Color
0x00FF0000	Pure blue
0x00000000	Black
0x00FFFFFF	White
0x00808080	Medium gray

The **RGB** macro accepts values for red, green, and blue, and returns an explicit RGB **COLORREF** value.

Palette Index

When specifying an index into a logical color palette, the **COLORREF** value has the following hexadecimal form:

```
0x0100iiii
```

The two low-order bytes consist of a 16-bit integer specifying an index into a logical palette. The third byte is not used and must be zero. The fourth (high-order) byte must be set to 1.

For example, the hexadecimal value 0x01000000 specifies the color in the palette entry of index 0; 0x0100000C specifies the color in the entry of index 12, and so on.

The **PALETTEINDEX** macro accepts an integer representing an index into a logical palette and returns a palette-index **COLORREF** value.

Palette-Relative RGB

When specifying a palette-relative RGB value, the **COLORREF** value has the following hexadecimal form:

```
0x02bbggrr
```

As with an explicit RGB, the three low-order bytes contain values for red, green, and blue; the high-order byte must be set to 2.

For output devices that support logical palettes, Windows matches a palette-relative RGB value to the nearest color in the logical palette of the device context, as though the application had specified an index to that palette entry. If an output device does not support a system palette, then Windows uses the palette-relative RGB as though it were an explict RGB **COLORREF** value.

The **PALETTERGB** macro accepts values for red, green, and blue, and returns a palette-relative RGB **COLORREF** value.

Comments

Before passing a palette-index or palette-relative RGB **COLORREF** value to a function that also requires a device-context parameter, an application that uses its own palette must select its palette into the device context (by calling the **SelectPalette** function) and realize the palette (by calling **RealizePalette**). This ensures that the function will use the correct palette-entry color. For functions that create an object (such as **CreatePen**), the application must select and realize the palette before selecting the object for the device context.

COMPAREITEMSTRUCT 3.0

Owner-Draw Item-Sorting Information

The **COMPAREITEMSTRUCT** structure supplies the identifiers and application-supplied data for two items in a sorted owner-draw combo box or list box.

Whenever an application adds a new item to an owner-draw combo or list box created with the CBS_SORT or LBS_SORT style, Windows sends the owner a WM_COMPAREITEM message. The *lParam* parameter of the message contains a long pointer to a **COMPARE-ITEMSTRUCT** data structure. When the owner receives the message, the owner compares the two items and returns a value indicating which item sorts before the other. For more information, see the description of the WM_COMPAREITEM message in Chapter 6, "Messages Directory," in *Reference, Volume 1.*

```
typedef struct tagCOMPAREITEMSTRUCT {
   WORD   CtlType;
   WORD   CtlID;
   HWND   hwndItem;
   WORD   itemID1;
   DWORD  itemData1;
   WORD   itemID2;
   DWORD  itemData2;
} COMPAREITEMSTRUCT;
```

The **COMPAREITEMSTRUCT** structure has the following fields:

Field	Description
CtlType	Is ODT_LISTBOX (which specifies an owner-draw list box) or ODT_COMBOBOX (which specifies an owner-draw combo box).
CtlID	Is the control ID for the list box or combo box.
hwndItem	Is the window handle of the control.
itemID1	Is the index of the first item in the list box or combo box being compared.
itemData1	Is application-supplied data for the first item being compared. This value was passed as the *lParam* parameter of the message that added the item to the combo or list box.

Field	Description
itemID2	Is the index of the second item in the list box or combo box being compared.
itemData2	Is application-supplied data for the second item being compared. This value was passed as the *lParam* parameter of the message that added the item to the combo or list box.

COMSTAT

Communication-Device Status

The **COMSTAT** structure contains information about a communications device.

```
typedef struct tagCOMSTAT {
   BYTE fCtsHold: 1;
   BYTE fDsrHold: 1;
   BYTE fRlsdHold: 1;
   BYTE fXoffHold: 1;
   BYTE fXoffSent: 1;
   BYTE fEof: 1;
   BYTE fTxim: 1;
   WORD cbInQue;
   WORD cbOutQue;
} COMSTAT;
```

The **COMSTAT** structure has the following fields:

Field	Description
fCtsHold: 1	Specifies whether transmission is waiting for the clear-to-send (CTS) signal to be sent.
fDsrHold: 1	Specifies whether transmission is waiting for the data-set-ready (DSR) signal to be sent.
fRlsdHold: 1	Specifies whether transmission is waiting for the receive-line-signal-detect (RLSD) signal to be sent.
fXoffHold: 1	Specifies whether transmission is waiting as a result of the **XoffChar** character being received.
fXoffSent: 1	Specifies whether transmission is waiting as a result of the **XoffChar** character being transmitted. Transmission halts when the **XoffChar** character is transmitted and used by systems that take the next character as XON, regardless of the actual character.

Field	Description
fEof: 1	Specifies whether the **EofChar** character has been received.
fTxim: 1	Specifies whether a character is waiting to be transmitted.
cbInQue	Specifies the number of characters in the receive queue.
cbOutQue	Specifies the number of characters in the transmit queue.

See Also The **GetCommError** function in Chapter 4, "Functions Directory," in *Reference, Volume 1.*

CREATESTRUCT

Window-Creation Structure

The **CREATESTRUCT** structure defines the initialization parameters passed to an application's window function.

```
typedef struct tagCREATESTRUCT {
   LPSTR   lpCreateParams;
   HANDLE  hInstance;
   HANDLE  hMenu;
   HWND    hwndParent;
   int     cy;
   int     cx;
   int     y;
   int     x;
   long    style;
   LPSTR   lpszName;
   LPSTR   lpszClass;
   long    ExStyle;
} CREATESTRUCT;
```

The **CREATESTRUCT** structure has the following fields:

Field	Description
lpCreateParams	Points to data to be used for creating the window.
hInstance	Identifies the module-instance handle of the module that owns the new window.
hMenu	Identifies the menu to be used by the new window.

Field	Description
hwndParent	Identifies the window that owns the new window. This field is NULL if the new window is a top-level window.
cy	Specifies the height of the new window.
cx	Specifies the width of the new window.
y	Specifies the *y*-coordinate of the upper-left corner of the new window. Coordinates are relative to the parent window if the new window is a child window. Otherwise, the coordinates are relative to the screen origin.
x	Specifies the *x*-coordinate of the upper-left corner of the new window. Coordinates are relative to the parent window if the new window is a child window. Otherwise, the coordinates are relative to the screen origin.
style	Specifies the new window's style.
lpszName	Points to a null-terminated character string that specifies the new window's name.
lpszClass	Points to a null-terminated character string that specifies the new window's class name.
ExStyle	Specifies extended style for the new window.

DCB

Communications-Device Control Block

The **DCB** structure defines the control setting for a serial communications device.

```
typedef struct tagDCB {
   BYTE Id;
   WORD BaudRate;
   BYTE ByteSize;
   BYTE Parity;
   BYTE StopBits;
   WORD RlsTimeout;
   WORD CtsTimeout;
   WORD DsrTimeout;

   BYTE fBinary: 1;
   BYTE fRtsDisable: 1;
   BYTE fParity: 1;
   BYTE fOutxCtsFlow: 1;
```

```
    BYTE fOutxDsrFlow: 1;
    BYTE fDummy: 2;
    BYTE fDtrDisable: 1;

    BYTE fOutX: 1;
    BYTE fInX: 1;
    BYTE fPeChar: 1;
    BYTE fNull: 1;
    BYTE fChEvt: 1;
    BYTE fDtrFlow: 1;
    BYTE fRtsFlow: 1;
    BYTE fDummy2: 1;

    char XonChar;
    char XoffChar;
    WORD XonLim;
    WORD XoffLim;
    char PeChar;
    char EofChar;
    char EvtChar;
    WORD TxDelay;
} DCB;
```

The **DCB** structure has the following fields:

Field	Description
Id	Specifies the communication device. This value is set by the device driver. If the most significant bit is set, then the **DCB** structure is for a parallel device.
BaudRate	Specifies the baud rate at which the communications device operates.
ByteSize	Specifies the number of bits in the characters transmitted and received. The **ByteSize** field can be any number from 4 to 8.
Parity	Specifies the parity scheme to be used. The **Parity** field can be any one of the following values:

Value	Meaning
EVENPARITY	Even
MARKPARITY	Mark
NOPARITY	No parity
ODDPARITY	Odd

Field	Description
	Value — **Meaning** SPACEPARITY — Space
StopBits	Specifies the number of stop bits to be used. The **StopBits** field can be any one of the following values:

Value	Meaning
ONESTOPBIT	1 stop bit
ONE5STOPBITS	1.5 stop bits
TWOSTOPBITS	2 stop bits

Field	Description
RlsTimeout	Specifies the maximum amount of time (in milliseconds) the device should wait for the receive-line-signal-detect (RLSD) signal. (RLSD is also known as the carrier detect (CD) signal.)
CtsTimeout	Specifies the maximum amount of time (in milliseconds) the device should wait for the clear-to-send (CTS) signal.
DsrTimeout	Specifies the maximum amount of time (in milliseconds) the device should wait for the data-set-ready (DSR) signal.
fBinary: 1	Specifies binary mode. In nonbinary mode, the **EofChar** character is recognized on input and remembered as the end of data.
fRtsDisable: 1	Specifies whether or not the request-to-send (RTS) signal is disabled. If the **fRtsDisable** field is set, RTS is not used and remains low. If **fRtsDisable** is clear, RTS is sent when the device is opened and turned off when the device is closed.
fParity: 1	Specifies whether parity checking is enabled. If the **fParity** field is set, parity checking is performed and errors are reported.
fOutxCtsFlow: 1	Specifies that clear-to-send (CTS) signal is to be monitored for output flow control. If the **fOutxCtsFlow** field is set and CTS is turned off, output is suspended until CTS is again sent.
fOutxDsrFlow: 1	Specifies that the data-set-ready (DSR) signal is to be monitored for output flow control. If the **fOutxDsrFlow** field is set and DSR is turned off, output is suspended until DSR is again sent.
fDummy: 2	Reserved.

Field	Description
fDtrDisable: 1	Specifies whether the data-terminal-ready (DTR) signal is disabled. If the **fDtrDisable** field is set, DTR is not used and remains low. If **fDtrDisable** is clear, DTR is sent when the device is opened and turned off when the device is closed.
fOutX: 1	Specifies that XON/XOFF flow control is used during transmission. If the **fOutX** field is set, transmission stops when the **XoffChar** character is received, and starts again when the **XonChar** character is received.
fInX: 1	Specifies that XON/XOFF flow control is used during reception. If the **fInX** field is set, the **XonChar** character is sent when the receive queue comes within **XoffLim** characters of being full, and the **XonChar** character is sent when the receive queue comes within **XonLim** characters of being empty.
fPeChar: 1	Specifies that characters received with parity errors are to be replaced with the character specified by the **fPeChar** field. The **fParity** field must be set for the replacement to occur.
fNull: 1	Specifies that received null characters are to be discarded.
fChEvt: 1	Specifies that reception of the **EvtChar** character is to be flagged as an event.
fDtrFlow: 1	Specifies that the data-terminal-ready (DTR) signal is to be used for receive flow control. If the **fDtrFlow** field is set, DTR is turned off when the receive queue comes within **XoffLim** characters of being full, and sent when the receive queue comes within **XonLim** characters of being empty.
fRtsFlow: 1	Specifies that the ready-to-send (RTS) signal is to be used for receive flow control. If the **fRtsFlow** field is set, RTS is turned off when the receive queue comes within **XoffLim** characters of being full, and sent when the receive queue comes within **XonLim** characters of being empty.
fdummy2: 1	Reserved.
XonChar	Specifies the value of the XON character for both transmission and reception.
XoffChar	Specifies the value of the XOFF character for both transmission and reception.
XonLim	Specifies the minimum number of characters allowed in the receive queue before the XON character is sent.

Field	Description
XoffLim	Specifies the maximum number of characters allowed in the receive queue before the XOFF character is sent. The **XoffLim** value is subtracted from the size of the receive queue (in bytes) to calculate the maximum number of characters allowed.
PeChar	Specifies the value of the character used to replace characters received with a parity error.
EofChar	Specifies the value of the character used to signal the end of data.
EvtChar	Specifies the value of the character used to signal an event.
TxDelay	Not currently used.

See Also The **BuildCommDCB**, **GetCommState**, and **SetCommState** functions in Chapter 4, "Functions Directory," in *Reference, Volume 1.*

DELETEITEMSTRUCT 3.0

Deleted Owner-Draw List-Box Item

The **DELETEITEMSTRUCT** structure describes a deleted owner-draw list-box or combo-box item. When an item is removed from the list box or combo box, or when the list box or combo box is destroyed, Windows sends the WM_DELETEITEM message to the owner for each deleted item; the *lParam* parameter of the message contains a pointer to this structure.

```
typedef struct tagDELETEITEMSTRUCT
  {
    WORD      CtlType
    WORD      CtlID;
    WORD      itemID;
    HWND      hwndItem;
    DWORD     itemData;
  } DELETEITEMSTRUCT;
```

The **DELETEITEMSTRUCT** structure has the following fields:

Field	Description
CtlType	Is ODT_LISTBOX (which specifies an owner-draw list box) or ODT_COMBOBOX (which specifies an owner-draw combo box).
CtlID	Is the control ID for the list box or combo box.
itemID	Is the index of the item in the list box or combo box being removed.
hwndItem	Is the window handle of the control.
itemData	Contains the value passed to the control in the *lParam* parameter of the LB_INSERTSTRING, LB_ADDSTRING, CB_INSERTSTRING, or CB_ADDSTRING message when the item was added to the list box.

DEVMODE 3.0

Printer Driver Initialization Information

The **DEVMODE** data structure contains information about the device initialization and environment of a printer driver. An application passes this structure to the **DeviceCapabilities** and **ExtDeviceMode** functions.

```
typedef struct _devicemode {
   char    dmDeviceName[32];
   WORD    dmSpecVersion;
   WORD    dmDriverVersion;
   WORD    dmSize;
   WORD    dmDriverExtra;
   DWORD   dmFields;
   short   dmOrientation;
   short   dmPaperSize;
   short   dmPaperLength;
   short   dmPaperWidth;
   short   dmScale;
   short   dmCopies;
   short   dmDefaultSource;
   short   dmPrintQuality;
   short   dmColor;
   short   dmDuplex;
   BYTE    dmDriverData[dmDriverExtra];
   } DEVMODE;
```

The **DEVMODE** structure contains the following fields:

Field	Description
dmDeviceName	Specifies the name of the device the driver supports; for example, "PCL/HP LaserJet" in the case of PCL/HP® LaserJet®. This string is unique among device drivers.
dmSpecVersion	Specifies the version number of the initialization data specification upon which the structure is based. The version number follows the Windows version number and is currently 0x300.
dmDriverVersion	Specifies the printer driver version number assigned by the printer driver developer.
dmSize	Specifies the size in bytes of the **DEVMODE** structure *except* the **dmDriverData** (device-specific) field. If an application manipulates only the driver-independent portion of the data, it can use this field to determine the length of the structure without having to account for different versions.
dmDriverExtra	Contains the size of the **dmDriverData** field and is the length of the device-specific data in the **DEVMODE** structure. If an application does not use device-specific information, it should set this field to zero.
dmFields	Is a bitfield that specifies which of the remaining fields in the **DEVMODE** structure have been initialized. Bit 0 (defined as DM_ORIENTATION) corresponds to **dmOrientation**; bit 1 (defined as DM_PAPERSIZE) specifies **dmPaperSize**, and so on. A printer driver supports only those fields that are appropriate for the printer technology.
dmOrientation	Selects the orientation of the paper. It can be either DMORIENT_PORTRAIT (1) or DMORIENT_LANDSCAPE (2).
dmPaperSize	Selects the size of the paper to print on. This field may be set to zero if the length and width of the paper are both set by the **dmPaperLength** and **dmPaperWidth** fields. Otherwise, the **dmPaperSize** field can be set to one of the following predefined values:

Value	Meaning
DMPAPER_LETTER	8½-by-11-inch paper

Field	Description	
	Value	**Meaning**
	DMPAPER_LEGAL	8½-by-14-inch paper
	DMPAPER_A4	210-by-297-millimeter paper
	DMPAPER_CSHEET	17-by-22-inch paper
	DMPAPER_DSHEET	22-by-34-inch paper
	DMPAPER_ESHEET	34-by-44-inch paper
	DMPAPER_ENV_9	3⅞-by-8⅞-inch #9 envelope
	DMPAPER_ENV_10	4⅛-by-9⅕-inch #10 envelope
	DMPAPER_ENV_11	4½-by-10⅜-inch #11 envelope
	DMPAPER_ENV_12	4¾-by-11-inch #12 envelope
	DMPAPER_ENV_14	5-by-11½-inch #14 envelope
dmPaperLength	Overrides the length of the paper specified by the **dmPaperSize** field, either for custom paper sizes or for devices such as dot-matrix printers which can print on a page of arbitrary length. These values, along with all other values which specify a physical length, are in tenths of a millimeter.	
dmPaperWidth	Overrides the width of the paper specified by the **dmPaperSize** field.	
dmScale	Scales the printed output. The apparent page size is scaled by a factor of **dmScale**/100 from the physical page size. A letter-size paper with a **dmScale** value of 50 would appear to be 17 by 22 inches, and output text and graphics would be correspondingly half their normal height and width.	
dmCopies	Selects the number of copies printed if the device supports multiple-page copies.	
dmDefaultSource	Specifies the paper bin from which the paper is fed by default. The application can override this selection by using the GETSETPAPERBINS escape. Possible bins include the following:	

Field	Description
	DMBIN_DEFAULT DMBIN_UPPER DMBIN_LOWER DMBIN_MANUAL DMBIN_TRACTOR DMBIN_ENVELOPE There is also a range of values reserved for device-specific bins. The GETSETPAPERBINS and ENUMPAPERBINS escapes use these indexes to be consistent with initialization information.
dmPrintQuality	Specifies the printer resolution. There are four predefined device-independent values: DMRES_HIGH (–4) DMRES_MEDIUM (–3) DMRES_LOW (–2) DMRES_DRAFT (–1) If a positive value is given, it specifies the number of dots per inch (DPI) and is therefore device dependent.
dmColor	Switches between color and monochrome on color printers. Possible values are: ■ DMCOLOR_COLOR (1) ■ DMCOLOR_MONOCHROME (2).
dmDuplex	Selects duplex or double-sided printing for printers capable of duplex printing. Values for this field include: ■ DMDUP_SIMPLEX (1) ■ DMDUP_HORIZONTAL (2) ■ DMDUP_VERTICAL (3).
dmDriverData[ ]	Contains device-specific data defined by the device driver.

Comments

Only drivers fully updated for Windows version 3.0 and which export the **ExtDeviceMode** function use the **DEVMODE** data structure.

DLGTEMPLATE

Dialog Template

The **DLGTEMPLATE** defines the contents of a dialog box. This structure is divided into three distinct parts:

Part	Description
Header Data Structure	Contains a general description of the dialog box.
Font-Information Data Structure	Defines the font with which text is drawn in the dialog box. This part is optional.
List of Items	Describes the parts that compose the dialog box.

The **CreateDialogIndirect**, **CreateDialogIndirectParam**, **DialogBoxIndirect**, and **DialogBoxIndirectParam** functions use this structure.

Header Data Structure

The **DLGTEMPLATE** header is shown here:

```
typedef struct {
    long  dtStyle;
    BYTE  dtItemCount;
    int   dtX;
    int   dtY;
    int   dtCX;
    int   dtCY;
    char  dtMenuName[];
    char  dtClassName[];
    char  dtCaptionText[];
} DLGTEMPLATE;
```

The **DLGTEMPLATE** header has the following fields:

Field	Description
dtStyle	Specifies the style of the dialog box. This field may be any or all of these values:

Field	Description	
	Value	**Meaning**
	DS_LOCALEDIT	Specifies that text storage for edit controls will be allocated in the application's local data segment. This allows the use of the EM_GETHANDLE and EM_SETHANDLE messages. If this style is not specified, edit-control data is located in a separate global data block.
	DS_SYSMODAL	Specifies a system-modal dialog box.
	DS_MODALFRAME	Specifies a dialog box with a modal dialog-box border. This style can be combined with the WS_CAPTION and WS_SYSMENU style flags to create a dialog box with a title bar and System menu.
	DS_ABSALIGN	Indicates that **dtX** and **dtY** are relative to the screen origin, not to the owner of the dialog box.
	DS_SETFONT	Specifies that a font other than the system font is to be used to draw text in the dialog box. If this flag is set, the **FONTINFO** data structure described in the following paragraphs must immediately follow the **DLGTEMPLATE** header. When Windows creates a dialog box with this attribute, Windows sends the WM_SETFONT message to the dialog-box window prior to creating the controls.

Field	Description	
	Value	**Meaning**
	DS_NOIDLEMSG	Specifies that Windows will not send the WM_ENTERIDLE message to the owner of the dialog box while the dialog box is displayed.
dtItemCount	Specifies the number of items in the dialog box. A dialog box can contain up to 255 controls.	
dtX	Specifies the *x*-coordinate of the upper-left corner of the dialog box in units of ¼ of the current dialog base width unit. The dialog base units are computed from the height and width of the current system font; the **GetDialogBaseUnits** function returns the current dialog base units in pixels. Unless DS_ABSALIGN is set in the **dtStyle** field, this value is relative to the origin of the parent window's client area.	
dtY	Specifies the *y*-coordinate of the upper-left corner of the dialog box in units of ⅛ of the current dialog base height unit. Unless DS_ABSALIGN is set in the **dtStyle** field, this value is relative to the origin of the parent window's client area.	
dtCX	Specifies the width of the dialog box in units of ¼ of the dialog base width unit.	
dtCY	Specifies the height of the dialog box in units of ⅛ of the dialog base height unit.	
dtMenuName[ ]	Specifies a null-terminated string that specifies the name of the dialog box's menu. If this field is NULL, the dialog-box window does not have a menu.	
dtClassName[ ]	Specifies a null-terminated string that supplies the name of the dialog box's class. If **dtClassName[ ]** is zero, it creates a dialog box with the standard dialog-box style. If an application specifies a class name, it should provide a dialog procedure that processes each dialog-box message directly or calls the **DefDlgProc** function to process the message. Also, the application must register the class with the **cbWndExtra** field of the **WNDCLASS** data structure set to **DLGWINDOWEXTRA**.	
dtCaptionText[ ]	Specifies a null-terminated string that supplies the caption for the dialog box.	

Font-Information Data Structure

The **FONTINFO** data structure contains information about the point size and face name of the font which Windows is to use to draw text in the dialog box.

```
typedef struct{
    short int   PointSize;
    char        szTypeFace[]; /* A null-terminated string */
} FONTINFO;
```

The **FONTINFO** structure has the following fields:

Field	Description
PointSize	Specifies the size of the typeface in points.
szTypeFace	Specifies the name of the typeface; for example, "Courier".

Comments

The font specified must have been previously loaded, either from WIN.INI or explicitly by calling the **LoadFont** function.

Item List

The item list consists of one or more **DLGITEMTEMPLATE** data structures, one for each control in the dialog box. The first such structure immediately follows the **FONTINFO** structure or the header at the first byte after the terminating null character in the **szTypeFace** field or the **dtCaptionText[ ]** field. The following shows the format of the **DLGITEMTEMPLATE** structure.

```
typedef struct {
    int   dtilX;
    int   dtilY;
    int   dtilCX;
    int   dtilCY;
    int   dtilID;
    long  dtilStyle;
    char  dtilClass[];
    char  dtilText[];
    BYTE  dtilInfo;
    PTR   dtilData;
} DLGITEMTEMPLATE
```

The **DLGITEMTEMPLATE** data structure has the following fields:

Field	Description
dtilX	Specifies the x-coordinate of the upper-left corner of the dialog-box item in units of 1/4 of the current dialog base width unit, relative to the origin of the dialog box. The dialog base units are computed from the height and width of the current system font. The **GetDialog-BaseUnits** function returns the current dialog base units in pixels.
dtilY	Specifies the y-coordinate of the upper-left corner of the dialog-box item in units of 1/8 of the current dialog base height unit. This value is relative to the origin of the dialog box.
dtilCX	Specifies the width-extent of the dialog-box item in units of 1/4 of the current dialog base width unit. Dialog base units are computed from the height and width of the current system font. The **GetDialog-BaseUnits** function returns the current dialog base units.
dtilCY	Specifies the height of the dialog-box item in units of 1/8 of the dialog base height unit.
dtilID	Specifies the dialog-box item identification number.
dtilStyle	Specifies the style of the dialog-box item.
dtilClass[ ]	A null-terminated string that specifies the control's class. It may be one of the following class names: BUTTON EDIT STATIC LISTBOX SCROLLBAR COMBOBOX
dtilText[ ]	Specifies the text for the item; it is a null-terminated string.
dtilInfo	Specifies the number of bytes of additional data that follows this item description and precedes the next item description.
dtilData	Specifies additional data which the **CreateWindow** function receives through the **lpCreateParams** field of the **CREATESTRUCT** data structure. This field is zero length if **dtilInfo** is zero.

DRAWITEMSTRUCT 3.0

Owner-Draw Control Drawing Information

The **DRAWITEMSTRUCT** structure provides information the owner needs to determine how to paint an owner-draw control. The owner of the owner-draw control receives a pointer to this structure as the *lParam* parameter of the WM_DRAWITEM message.

```
typedef struct tagDRAWITEMSTRUCT
  {
    WORD   CtlType;
    WORD   CtlID;
    WORD   itemID;
    WORD   itemAction;
    WORD   itemState;
    HWND   hwndItem;
    HDC    hDC;
    RECT   rcItem;
    DWORD  itemData;
  } DRAWITEMSTRUCT;
```

The **DRAWITEMSTRUCT** structure has the following fields:

Field	Description
CtlType	Is the control type. The values for control types are as follows:

Value	Meaning
ODT_BUTTON	Owner-draw button.
ODT_COMBOBOX	Owner-draw combo box.
ODT_LISTBOX	Owner-draw list box.
ODT_MENU	Owner-draw menu.

Field	Description
CtlID	Is the control ID for a combo box, list box or button. This field is not used for a menu.
itemID	Is the menu-item ID for a menu or the index of the item in a list box or combo box. For an empty list box or combo box, this field can be –1. This allows the application to draw only the focus rectangle at the coordinates specified by the **rcItem** field even though there are no items in the control. This indicates to the user whether the list box or combo box has input focus. The setting of the bits in the **itemAction** field determines whether the rectangle is to be drawn as though the list box or combo box has input focus.

Field	Description
itemAction	Defines the drawing action required. This will be one or more of the following bits:
itemState	Specifies the visual state of the item *after* the current drawing action takes place. That is, if a menu item is to be grayed, the state flag ODS_GRAYED will be set. The state flags are:
hwndItem	For combo boxes, list boxes and buttons, this field specifies the window handle of the control; for menus, it contains the handle of the menu (**HMENU**) containing the item.

itemAction values:

Value	Description
ODA_DRAWENTIRE	This bit is set when the entire control needs to be drawn.
ODA_FOCUS	This bit is set when the control gains or loses input focus. The **itemState** field should be checked to determine whether the control has focus.
ODA_SELECT	This bit is set when only the selection status has changed. The **itemState** field should be checked to determine the new selection state.

itemState values:

Value	Description
ODS_CHECKED	This bit is set if the menu item is to be checked. This bit is used only in a menu.
ODS_DISABLED	This bit is set if the item is to be drawn as disabled.
ODS_FOCUS	This bit is set if the item has input focus.
ODS_GRAYED	This bit is set if the item is to be grayed. This bit is used only in a menu.
ODS_SELECTED	This bit is set if the item's status is selected.

Field	Description
hDC	Identifies a device context; this device context must be used when performing drawing operations on the control.
rcItem	Is a rectangle in the device context specified by the **hDC** field that defines the boundaries of the control to be drawn. Windows automatically clips anything the owner draws in the device context for combo boxes, list boxes, and buttons, but does not clip menu items. When drawing menu items, the owner must ensure that the owner does not draw outside the boundaries of the rectangle defined by the **rcItem** field.
itemData	For a combo box or list box, this field contains the value that was passed to the list box in the *lParam* parameter of one of the the following messages: ■ CB_ADDSTRING ■ CB_INSERTSTRING ■ LB_ADDSTRING ■ LB_INSERTSTRING For a menu, this field contains the **DWORD** value passed as the *lpNewItem* parameter of the **InsertMenu** which inserted the menu item. Its contents are undefined for buttons.

HANDLETABLE

Window-Handle Table

The **HANDLETABLE** structure is an array of handles, each of which identifies a GDI object.

```
HANDLE objectHandle[1]
```

The **HANDLETABLE** structure has the following field:

Field	Description
objectHandle[1]	Identifies an array of handles.

LOGBRUSH

Logical-Brush Attribute Information

The **LOGBRUSH** structure defines the style, color, and pattern of a physical brush to be created by using the **CreateBrushIndirect** function.

```
typedef struct tagLOGBRUSH {
   WORD      lbStyle;
   COLORREF  lbColor;
   short int lbHatch;
} LOGBRUSH;
```

The **LOGBRUSH** structure has the following fields:

Field	Description
lbStyle	Specifies the brush style. The **lbStyle** field can be any one of the following styles:

Style	Meaning
BS_DIBPATTERN	Specifies a pattern brush defined by a device-independent bitmap (DIB) specification.
BS_HATCHED	Specifies a hatched brush.
BS_HOLLOW	Specifies a hollow brush.
BS_PATTERN	Specifies a pattern brush defined by a memory bitmap.
BS_SOLID	Specifies a solid brush.

Field	Description
lbColor	Specifies the color in which the brush is to be drawn. If **lbStyle** is BS_HOLLOW or BS_PATTERN, **lbColor** is ignored.

If **lpStyle** is BS_DIBPATTERN, the low-order word of **lbColor** specifies whether the **bmiColors** fields of the **BITMAPINFO** data structure contain explicit RGB values or indexes into the currently realized logical palette. The **lbColor** field must be one of the following values:

Value	Meaning
DIB_PAL_COLORS	The color table consists of an array of 16-bit indexes into the currently realized logical palette.
DIB_RGB_COLORS	The color table contains literal RGB values.

Field	Description
lbHatch	Specifies a hatch style. The meaning depends on the brush style.

If **lbStyle** is BS_DIBPATTERN, the **lbHatch** field contains a handle to a packed DIB. To obtain this handle, an application calls the **GlobalAlloc** function to allocate a block of global memory and then fills the memory with the packed DIB. A packed DIB consists of a **BITMAPINFO** data structure immediately followed by the array of bytes which define the pixels of the bitmap.

If **lbStyle** is BS_HATCHED, the **lbHatch** field specifies the orientation of the lines used to create the hatch. It can be any one of the following values:

Value	Meaning
HS_BDIAGONAL	45-degree upward hatch (left to right)
HS_CROSS	Horizontal and vertical crosshatch
HS_DIAGCROSS	45-degree crosshatch
HS_FDIAGONAL	45-degree downward hatch (left to right)
HS_HORIZONTAL	Horizontal hatch
HS_VERTICAL	Vertical hatch

If **lbStyle** is BS_PATTERN, **lbHatch** must be a handle to the bitmap that defines the pattern.

If **lbStyle** is BS_SOLID or BS_HOLLOW, **lbHatch** is ignored.

See Also The **CreateBrushIndirect** function in Chapter 4, "Functions Directory," in *Reference, Volume 1.*

LOGFONT

Logical-Font Descriptor

The **LOGFONT** structure defines the attributes of a font, a drawing object used to write text on a display surface.

```
typedef struct tagLOGFONT {
   short int lfHeight;
   short int lfWidth;
   short int lfEscapement;
   short int lfOrientation;
   short int lfWeight;
   BYTE      lfItalic;
   BYTE      lfUnderline;
   BYTE      lfStrikeOut;
   BYTE      lfCharSet;
   BYTE      lfOutPrecision;
   BYTE      lfClipPrecision;
   BYTE      lfQuality;
   BYTE      lfPitchAndFamily;
   BYTE      lfFaceName[LF_FACESIZE];
} LOGFONT;
```

The **LOGFONT** structure has the following fields:

Field	Description
lfHeight	Specifies the average height of the font (in user units). The height of a font can be specified in the following three ways. If the **lfHeight** field is greater than zero, it is transformed into device units and matched against the cell height of the available fonts. If **lfHeight** is zero, a reasonable default size is used. If **lfHeight** is less than zero, it is transformed into device units and the absolute value is matched against the character height of the available fonts.
lfWidth	Specifies the average width of characters in the font (in device units). If the **lfWidth** field is zero, the aspect ratio of the device is matched against the digitization aspect ratio of the available fonts for the closest match by absolute value of the difference.
lfEscapement	Specifies the angle (in tenths of degrees) between the escapement vector and the *x*-axis of the display surface. The escapement vector is the line through the origins of the first and last characters on a line. The angle is measured counterclockwise from the *x*-axis.
lfOrientation	Specifies the angle (in tenths of degrees) between the baseline of a character and the *x*-axis. The angle is measured counterclockwise from the *x*-axis.

Field	Description
lfWeight	Specifies the font weight (in inked pixels per 1000). Although the **lfWeight** field can be any integer value from 0 to 1000, the common values are as follows: 400 Normal 700 Bold These values are approximate; the actual appearance depends on the font face. If **lfWeight** is zero, a default weight is used.
lfItalic	Specifies an italic font if set to nonzero.
lfUnderline	Specifies an underlined font if set to nonzero.
lfStrikeOut	Specifies a strikeout font if set to nonzero.
lfCharSet	Specifies the font's character set. The three values are predefined: ANSI_CHARSET OEM_CHARSET SYMBOL_CHARSET The OEM character set is system-dependent. Fonts with other character sets may exist in the system. If an application uses a font with an unknown character set, it should not attempt to translate or interpret strings that are to be rendered with that font. Instead, the strings should be passed directly to the output device driver.
lfOutPrecision	Specifies the font's output precision, which defines how closely the output must match the requested font's height, width, character orientation, escapement, and pitch. The default setting is OUT_DEFAULT_PRECIS.
lfClipPrecision	Specifies the font's clipping precision, which defines how to clip characters that are partially outside the clipping region. The default setting is CLIP_DEFAULT_PRECIS.
lfQuality	Specifies the font's output quality, which defines how carefully GDI must attempt to match the logical-font attributes to those of an actual physical font. It can be any one of the following values:

Field	Description	
	Value	**Meaning**
	DEFAULT_QUALITY	Appearance of the font does not matter.
	DRAFT_QUALITY	Appearance of the font is less important than when PROOF_QUALITY is used. For GDI fonts, scaling is enabled, which means that more font sizes are available, but the quality may be lower. Bold, italic, underline, and strikeout fonts are synthesized if necessary.
	PROOF_QUALITY	Character quality of the font is more important than exact matching of the logical-font attributes. For GDI fonts, scaling is disabled and the font closest in size is chosen. Although the chosen font size may not be mapped exactly when PROOF_QUALITY is used, the quality of the font is high and there is no distortion of appearance. Bold, italic, underline, and strikeout fonts are synthesized if necessary.

lfPitchAndFamily Specifies the font pitch and family. The two low-order bits specify the pitch of the font and can be any one of the following values:

DEFAULT_PITCH
FIXED_PITCH
VARIABLE_PITCH

The four high-order bits of the field specify the font family and can be any one of the following values:

Field	Description
	FF_DECORATIVE FF_DONTCARE FF_MODERN FF_ROMAN FF_SCRIPT FF_SWISS
	The proper value can be obtained by using the Boolean OR operator to join one pitch constant with one family constant.
	Font families describe the look of a font in a general way. They are intended for specifying fonts when the exact typeface desired is not available. The values for font families are as follows:

Value	Meaning
FF_DECORATIVE	Novelty fonts.
FF_DONTCARE	Don't care or don't know.
FF_MODERN	Fonts with constant stroke width (fixed-pitch), with or without serifs. Fixed-pitch fonts are usually modern.
FF_ROMAN	Fonts with variable stroke width (proportionally spaced) and with serifs.
FF_SCRIPT	Fonts designed to look like handwriting.
FF_SWISS	Fonts with variable stroke width (proportionally spaced) and without serifs.

Field	Description
lfFaceName	Specifies the font's typeface. It must be a null-terminated character string. If **lfFaceName** is NULL, GDI uses a default typeface.

See Also The **CreateFontIndirect** function in Chapter 4, "Functions Directory," in *Reference, Volume 1.*

LOGPALETTE 3.0

Logical Color Palette Information

The **LOGPALETTE** data structure defines a logical color palette.

```
typedef struct
        {
        WORD            palVersion;
        WORD            palNumEntries;
        PALETTEENTRY    palPalEntry[];
        } LOGPALETTE;
```

The **LOGPALETTE** structure has the following fields:

Field	Description
palVersion	Specifies the Windows version number for the structure (currently 0x300).
palNumEntries	Specifies the number of palette color entries.
palPalEntry [ ]	Specifies an array of **PALETTEENTRY** data structures that define the color and usage of each entry in the logical palette.

Comments The colors in the palette entry table should appear in order of importance. This is because entries earlier in the logical palette are most likely to be placed in the system palette.

This data structure is passed as a parameter to the **CreatePalette** function.

LOGPEN

Logical-Pen Attribute Information

The **LOGPEN** structure defines the style, width, and color of a pen, a drawing object used to draw lines and borders. The **CreatePenIndirect** function uses the **LOGPEN** structure.

```
typedef struct tagLOGPEN {
   WORD      lopnStyle;
   POINT     lopnWidth;
   COLORREF  lopnColor;
} LOGPEN;
```

The **LOGPEN** structure has the following fields:

Field	Description
lopnStyle	Specifies the pen type, which can be any one of the following values:

Constant Name	Value	Result
PS_SOLID	0	___
PS_DASH	1	---
PS_DOT	2	....
PS_DASHDOT	3	-.-.
PS_DASHDOTDOT	4	-..-..
PS_NULL	5	
PS_INSIDEFRAME	6	___

If the width of the pen is greater than 1 and the pen style is PS_INSIDEFRAME, the line is drawn inside the frame of all primitives except polygons and polylines; the pen is drawn with a logical (dithered) color if the pen color does not match an available RGB value. The PS_INSIDEFRAME style is identical to PS_SOLID if the pen width is less than or equal to 1.

Field	Description
lopnWidth	Specifies the pen width (in logical units). If the **lopnWidth** field is zero, the pen is one pixel wide on raster devices.
lopnColor	Specifies the pen color.

Comments

The *y* value in the **POINT** structure for **lopnWidth** is not used.

See Also

The **CreatePenIndirect** function in Chapter 4, "Functions Directory," in *Reference, Volume 1.*

MDICREATESTRUCT 3.0

MDI Child Window Creation Structure

The **MDICREATESTRUCT** data structure contains information about the class, title, owner, location, and size of a multiple document interface (MDI) child window.

```
typedef struct tagMDICREATESTRUCT
  {
    LPSTR   szClass;
    LPSTR   szTitle;
    HANDLE  hOwner;
    int     x;
    int     y;
    int     cx;
    int     cy;
    LONG    style;
    LONG    lParam;
  } MDICREATESTRUCT;
```

The **MDICREATESTRUCT** structure contains the following fields:

Field	Description
szClass	Contains a long pointer to the application-defined class of the MDI child window.
szTitle	Contains a long pointer to the window title of the MDI child window.
hOwner	Is the instance handle of the application creating the MDI child window.
x	Specifies the initial position of the left side of the MDI child window. If set to CW_USEDEFAULT, the MDI child window is assigned a default horizontal position.
y	Specifies the initial position of the top edge of the MDI child window. If set to CW_USEDEFAULT, the MDI child window is assigned a default vertical position.
cx	Specifies the initial width of the MDI child window. If set to CW_USEDEFAULT, the MDI child window is assigned a default width.
cy	Specifies the initial height of the MDI child window. If set to CW_USEDEFAULT, the MDI child window is assigned a default height.

Field	Description
style	Specifies additional styles for the MDI child window. The **style** field may be set to one or more of the following values:
	Value / **Meaning**
	WS_MINIMIZE: The MDI child window is created in a minimized state.
	WS_MAXIMIZE: The MDI child window is created in a maximized state.
	WS_HSCROLL: The MDI child window is created with a horizontal scroll bar.
	WS_VSCROLL: The MDI child window is created with a vertical scroll bar.
lParam	Is an application-defined 32-bit value.

Comments

When the MDI child window is created, Windows sends the WM_CREATE message to the window. The *lParam* parameter of the WM_CREATE message contains a pointer to a **CREATESTRUCT** data structure. The **lpCreateParams** field of the **CREATESTRUCT** structure contains a pointer to the **MDICREATESTRUCT** data structure passed with the WM_MDICREATE message that created the MDI child window.

MEASUREITEMSTRUCT 3.0

Owner-Draw Control Dimensions

The **MEASUREITEMSTRUCT** data structure informs Windows of the dimensions of an owner-draw control. This allows Windows to process user interaction with the control correctly. The owner of an owner-draw control receives a pointer to this structure as the *lParam* parameter of an WM_MEASUREITEM message. The owner-draw control sends this message to its owner window when the control is created; the owner then fills in the appropriate fields in the structure for the control and returns. This structure is common to all owner-draw controls.

The **MEASUREITEMSTRUCT** structure has the following format:

```
typedef struct tagMEASUREITEMSTRUCT
  {
    WORD   CtlType;
    WORD   CtlID;
    WORD   itemID;
    WORD   itemWidth;
```

```
    WORD   itemHeight;
    DWORD  itemData
} MEASUREITEMSTRUCT;
```

The **MEASUREITEMSTRUCT** structure contains the following fields:

Field	Description
CtlType	Is the control type. The values for control types are as follows: **Value** — **Meaning** ODT_BUTTON — Owner-draw button. ODT_COMBOBOX — Owner-draw combo box. ODT_LISTBOX — Owner-draw list box. ODT_MENU — Owner-draw menu.
CtlID	Is the control ID for a combo box, list box, or button. This field is not used for a menu.
itemID	Is the menu-item ID for a menu or the list-box item ID for a variable-height combo box or list box. This field is not used for a fixed-height combo box or list box, or for a button.
itemWidth	Specifies the width of a menu item. The owner of the owner-draw menu item must fill this field before returning from the message.
itemHeight	Specifies the height of an individual item in a list box or a menu. Before returning from the message, the owner of the owner-draw combo box, list box, or menu item must fill out this field.
itemData	Contains the value that was passed to the combo box or list box in the *lParam* parameter of one of the following messages: ■ CB_ADDSTRING ■ CB_INSERTSTRING ■ LB_ADDSTRING ■ LB_INSERTSTRING Contains the **DWORD** value passed as the *lpNewItem* parameter of the **AppendMenu**, **InsertMenu**, or **ModifyMenu** function that added or modified the menu item. Its contents are undefined for buttons.

Comments Failure to fill out the proper fields in the **MEASUREITEM** structure will cause improper operation of the control.

MENUITEMTEMPLATE

Menu-ItemTemplate

A complete menu template consists of a header and one or more menu-item lists. The following shows the structure of the menu-template header:

```
typedef struct {
   WORD   versionNumber;
   WORD   offset;
} MENUITEMTEMPLATEHEADER;
```

The menu-template header contains the following fields:

Field	Description
versionNumber	Specifies the version number. Should be zero.
offset	Specifies the offset from the header in bytes where the menu-item list begins.

One or more **MENUITEMTEMPLATE** structures are combined to form the menu-item list.

```
typedef struct {
   WORD mtOption;
   WORD mtID;
   LPSTR mtString;
} MENUITEMTEMPLATE;
```

The **MENUITEMTEMPLATE** structure has the following fields:

Field	Description
mtOption	Specifies a mask of one or more predefined menu options that specify the appearance of the menu item. The menu options are as follows:

Field	Description	
	Value	**Meaning**
	MF_CHECKED	Item has a checkmark next to it.
	MF_END	Item must be specified for the last item in a pop-up menu or a static menu.
	MF_GRAYED	Item is initially inactive and drawn with a gray effect.
	MF_HELP	Item has a vertical separator to its left.
	MF_MENUBARBREAK	Item is placed in a new column. The old and new columns are separated by a bar.
	MF_MENUBREAK	Item is placed in a new column.
	MF_OWNERDRAW	The owner of the menu is responsible for drawing all visual aspects of the menu item, including highlighted, checked and inactive states. This option is not valid for a top-level menu item.
	MF_POPUP	Item displays a sublist of menu items when selected.
mtID	Specifies an identification code for a nonpop-up menu item. The **MENUITEMTEMPLATE** data structure for a pop-up menu item does not contain the **mtID** field.	
mtString	Points to a null-terminated character string that specifies the name of the menu item.	

See also The **LoadMenuIndirect** function in Chapter 4, "Functions Directory," in *Reference, Volume 1.*

METAFILEPICT

Metafile Picture Structure

The **METAFILEPICT** structure defines the metafile picture format used for exchanging metafile data through the clipboard.

```
typedef struct tagMETAFILEPICT {
   int     mm;
   int     xExt, yExt;
   HANDLE  hMF;
} METAFILEPICT;
```

The **METAFILEPICT** structure has the following fields:

Field	Description
mm	Specifies the mapping mode in which the picture is drawn.
xExt	Specifies the size of the metafile picture for all modes except the MM_ISOTROPIC and MM_ANISOTROPIC modes. The *x*-extent specifies the width of the rectangle within which the picture is drawn. The coordinates are in units that correspond to the mapping mode.
yExt	Specifies the size of the metafile picture for all modes except the MM_ISOTROPIC and MM_ANISOTROPIC modes. The *y*-extent specifies the height of the rectangle within which the picture is drawn. The coordinates are in units that correspond to the mapping mode. For MM_ISOTROPIC and MM_ANISOTROPIC modes, which can be scaled, the **xExt** and **yExt** fields contain an optional suggested size in MM_HIMETRIC units. For MM_ANISOTROPIC pictures, **xExt** and **yExt** can be zero when no suggested size is supplied. For MM_ISOTROPIC pictures, an aspect ratio must be supplied even when no suggested size is given. (If a suggested size is given, the aspect ratio is implied by the size.) To give an aspect ratio without implying a suggested size, set **xExt** and **yExt** to negative values whose ratio is the appropriate aspect ratio. The magnitude of the negative **xExt** and **yExt** values will be ignored; only the ratio will be used.
hMF	Identifies a memory metafile.

MSG

Message Data Structure

The **MSG** structure contains information from the Windows application queue.

```
typedef struct tagMSG {
   HWND    hwnd;
   WORD    message;
   WORD    wParam;
   LONG    lParam;
   DWORD   time;
   POINT   pt;
} MSG;
```

The **MSG** structure has the following fields:

Field	Description
hwnd	Identifies the window that receives the message.
message	Specifies the message number.
wParam	Specifies additional information about the message. The exact meaning depends on the **message** value.
lParam	Specifies additional information about the message. The exact meaning depends on the **message** value.
time	Specifies the time at which the message was posted.
pt	Specifies the position of the cursor (in screen coordinates) when the message was posted.

MULTIKEYHELP

Windows Help Key Word Table Structure

The **MULTIKEYHELP** structure specifies a key-word table and an associated key word to be used by the Windows Help application.

```
typedef struct tagMULTIKEYHELP {
   WORD  mkSize;
   BYTE  mkKeylist;
   BYTE  szKeyphrase[];
} MULTIKEYHELP;
```

The **MULTIKEYHELP** data structure contains the following fields:

Field	Description
mkSize	Specifies the length of the **MULTIKEYHELP** structure (in bytes).
mkKeylist	Contains a single character that identifies the key-word table to be searched.
szKeyphrase[ ]	Contains a null-terminated text string that specifies the key word to be located in the key-word table.

OFSTRUCT

Open-File Structure

The **OFSTRUCT** structure contains file information which results from opening that file.

```
typedef struct tagOFSTRUCT {
   BYTE  cBytes;
   BYTE  fFixedDisk;
   WORD  nErrCode;
   BYTE  reserved[4];
   BYTE  szPathName[120];
} OFSTRUCT;
```

The **OFSTRUCT** structure has the following fields:

Field	Description
cBytes	Specifies the length of the **OFSTRUCT** structure (in bytes).
fFixedDisk	Specifies whether the file is on a fixed disk. The **fFixedDisk** field is nonzero if the file is on a fixed disk.
nErrCode	Specifies the DOS error code if the **OpenFile** function returns –1 (that is, **OpenFile** failed).
reserved[4]	Reserved field. Four bytes reserved for future use.
szPathName[120]	Specifies 120 bytes that contain the pathname of the file. This string consists of characters from the OEM character set.

PAINTSTRUCT

Windows Paint Information

The **PAINTSTRUCT** structure contains information for an application. This information can be used to paint the client area of a window owned by that application.

```
typedef struct tagPAINTSTRUCT {
   HDC  hdc;
   BOOL fErase;
   RECT rcPaint;
   BOOL fRestore;
   BOOL fIncUpdate;
   BYTE rgbReserved[16];
} PAINTSTRUCT;
```

The **PAINTSTRUCT** structure has the following fields:

Field	Description
hdc	Identifies the display context to be used for painting.
fErase	Specifies whether the background has been redrawn. It has been redrawn if nonzero.
rcPaint	Specifies the upper-left and lower-right corners of the rectangle in which the painting is requested.
fRestore	Reserved field. It is used internally by Windows.
fIncUpdate	Reserved field. It is used internally by Windows.
rgbReserved[16]	Reserved field. A reserved block of memory used internally by Windows.

PALETTEENTRY 3.0

Logical Palette Color Entry

The **PALETTEENTRY** data structure specifies the color and usage of an entry in a logical color palette. A logical palette is defined by a **LOGPALETTE** data structure.

```
typedef struct
        {
        BYTE  peRed;
        BYTE  peGreen;
        BYTE  peBlue;
        BYTE  peFlags;
        } PALETTEENTRY;
```

The **PALETTEENTRY** structure contains the following fields:

Field	Description
peRed	Specifies the intensity of red for the palette entry color.
peGreen	Specifies the intensity of green for the palette entry color.
peBlue	Specifies the intensity of blue for the palette entry color.
peFlags	Specifies how the palette entry is to be used. The **peFlags** field may be set to NULL or one of these values:

Flag	Meaning
PC_EXPLICIT	Specifies that the low-order word of the logical palette entry designates a hardware palette index. This flag allows the application to show the contents of the display-device palette.
PC_NOCOLLAPSE	Specifies that the color will be placed in an unused entry in the system palette instead of being matched to an existing color in the system palette. If there are no unused entries in the system palette, the color is matched normally. Once this color is in the system palette, colors in other logical palettes can be matched to this color.
PC_RESERVED	Specifies that the logical palette entry will be used for palette animation; this prevents other windows from matching colors to this palette entry since the color will frequently change. If an unused system-palette entry is available, this color is placed in that entry. Otherwise, the color will not be available for animation.

POINT

Point Data Structure

The **POINT** structure defines the *x*- and *y*-coordinates of a point.

```
typedef struct tagPOINT {
   int x;
   int y;
} POINT;
```

The **POINT** structure has the following fields:

Field	Description
x	Specifies the *x*-coordinate of a point.
y	Specifies the *y*-coordinate of a point.

See Also The **ChildWindowFromPoint**, **PtInRect**, and **WindowFromPoint** functions in Chapter 4, "Functions Directory," in *Reference, Volume 1.*

RECT

Rectangle Data Structure

The **RECT** structure defines the coordinates of the upper-left and lower-right corners of a rectangle.

```
typedef struct tagRECT {
   int left;
   int top;
   int right;
   int bottom;
} RECT;
```

The **RECT** structure has the following fields:

Field	Description
left	Specifies the *x*-coordinate of the upper-left corner of a rectangle.
top	Specifies the *y*-coordinate of the upper-left corner of a rectangle.
right	Specifies the *x*-coordinate of the lower-right corner of a rectangle.
bottom	Specifies the *y*-coordinate of the lower-right corner of a rectangle.

Comments The width of the rectangle defined by the **RECT** structure must not exceed 32,768 units.

RGBQUAD 3.0

RGB Color Structure

The **RGBQUAD** data structure describes a color consisting of relative intensities of red, green, and blue. The **bmiColors** field of the **BITMAPINFO** data structure consists of an array of **RGBQUAD** data structures.

```
typedef struct tagRGBQUAD {
   BYTE    rgbBlue;
   BYTE    rgbGreen;
   BYTE    rgbRed;
   BYTE    rgbReserved;
} RGBQUAD;
```

The **RGBQUAD** structure contains the following fields:

Field	Description
rgbBlue	Specifies the intensity of blue in the color.
rgbGreen	Specifies the intensity of green in the color.
rgbRed	Specifies the intensity of red in the color.
rgbReserved	Is not used and must be set to zero.

RGBTRIPLE 3.0

RGB Color Structure

The **RGBTRIPLE** data structure describes a color consisting of relative intensities of red, green, and blue. The **bmciColors** field of the **BITMAPCOREINFO** data structure consists of an array of **RGBTRIPLE** data structures.

```
typedef struct tagRGBTRIPLE {
        BYTE    rgbtBlue;
        BYTE    rgbtGreen;
        BYTE    rgbtRed;
} RGBTRIPLE;
```

The **RGBTRIPLE** structure contains the following fields:

Field	Description
rgbtBlue	Specifies the intensity of blue in the color.
rgbtGreen	Specifies the intensity of green in the color.
rgbtRed	Specifies the intensity of red in the color.

TEXTMETRIC

Basic Font Metrics

The **TEXTMETRIC** structure contains basic information about a physical font. All sizes are given in logical units; that is, they depend on the current mapping mode of the display context.

```
typedef struct tagTEXTMETRIC {
   short int tmHeight;
   short int tmAscent;
   short int tmDescent;
   short int tmInternalLeading;
   short int tmExternalLeading;
   short int tmAveCharWidth;
   short int tmMaxCharWidth;
   short int tmWeight;
   BYTE      tmItalic;
   BYTE      tmUnderlined;
   BYTE      tmStruckOut;
   BYTE      tmFirstChar;
   BYTE      tmLastChar;
   BYTE      tmDefaultChar;
   BYTE      tmBreakChar;
   BYTE      tmPitchAndFamily;
   BYTE      tmCharSet;
   short int tmOverhang;
   short int tmDigitizedAspectX;
   short int tmDigitizedAspectY;
} TEXTMETRIC;
```

The **TEXTMETRIC** structure has the following fields:

Field	Description
tmHeight	Specifies the height (ascent + descent) of characters.
tmAscent	Specifies the ascent (units above the baseline) of characters.

Field	Description
tmDescent	Specifies the descent (units below the baseline) of characters.
tmInternalLeading	Specifies the amount of leading (space) inside the bounds set by the **tmHeight** field. Accent marks and other foreign characters may occur in this area. The designer may set this field to zero.
tmExternalLeading	Specifies the amount of extra leading (space) that the application adds between rows. Since this area is outside the font, it contains no marks and will not be altered by text output calls in either OPAQUE or TRANSPARENT mode. The designer may set this field to zero.
tmAveCharWidth	Specifies the average width of characters in the font (loosely defined as the width of the letter *x*). This value does not include overhang required for bold or italic characters.
tmMaxCharWidth	Specifies the width of the widest character in the font.
tmWeight	Specifies the weight of the font.
tmItalic	Specifies an italic font if it is nonzero.
tmUnderlined	Specifies an underlined font if it is nonzero.
tmStruckOut	Specifies a struckout font if it is nonzero.
tmFirstChar	Specifies the value of the first character defined in the font.
tmLastChar	Specifies the value of the last character defined in the font.
tmDefaultChar	Specifies the value of the character that will be substituted for characters that are not in the font.
tmBreakChar	Specifies the value of the character that will be used to define word breaks for text justification.
tmPitchAndFamily	Specifies the pitch and family of the selected font. The low-order bit specifies the pitch of the font. If it is 1, the font is variable pitch. If it is 0, the font is fixed pitch.

Field	Description
	The four high-order bits designate the font family. The **tmPitchAndFamily** field can be combined with the hexadecimal value 0xF0 by using the bitwise AND operator, and then be compared with the font family names for an identical match. For a description of the font families, see the **LOGFONT** structure, earlier in this chapter.
tmCharSet	Specifies the character set of the font.
tmOverhang	Specifies the per-string extra width that may be added to some synthesized fonts. When synthesizing some attributes, such as bold or italic, GDI or a device may have to add width to a string on both a per-character and per-string basis. For example, GDI makes a string bold by expanding the intracharacter spacing and overstriking by an offset value; it italicizes a font by skewing the string. In either case, there is an overhang past the basic string. For bold strings, the overhang is the distance by which the overstrike is offset. For italic strings, the overhang is the amount the top of the font is skewed past the bottom of the font. The **tmOverhang** field allows the application to determine how much of the character width returned by a **GetTextExtent** function call on a single character is the actual character width and how much is the per-string extra width. The actual width is the extent minus the overhang.
tmDigitizedAspectX	Specifies the horizontal aspect of the device for which the font was designed.
tmDigitizedAspectY	Specifies the vertical aspect of the device for which the font was designed. The ratio of the **tmDigitizedAspectX** and **tmDigitizedAspectY** fields is the aspect ratio of the device for which the font was designed.

See Also

The **GetDeviceCaps** and **GetTextMetrics** functions in Chapter 4, "Functions Directory," in *Reference, Volume 1.*

WNDCLASS

Window-Class Data Structure

The **WNDCLASS** structure contains the class attributes that are registered by the **RegisterClass** function.

```
typedef struct tagWNDCLASS {
   WORD    style;
   long    (FAR PASCAL *lpfnWndProc)();
   int     cbClsExtra;
   int     cbWndExtra;
   HANDLE  hInstance;
   HICON   hIcon;
   HCURSOR hCursor;
   HBRUSH  hbrBackground;
   LPSTR   lpszMenuName;
   LPSTR   lpszClassName;
} WNDCLASS;
```

The **WNDCLASS** structure has the following fields:

Field	Description
style	Specifies the class style. These styles can be combined by using the bitwise OR operator. The **style** field can be any combination of the following values:

Value	Meaning
CS_BYTEALIGNCLIENT	Aligns a window's client area on the byte boundary (in the *x* direction).
CS_BYTEALIGNWINDOW	Aligns a window on the byte boundary (in the *x* direction).
CS_CLASSDC	Gives the window class its own display context (shared by instances).
CS_DBLCLKS	Sends double-click messages to a window.

Field	Description	
	Value	**Meaning**
	CS_GLOBALCLASS	Specifies that the window class is an application global class. An application global class is created by an application or library and is available to all applications. The class is destroyed when the application or library that created the class terminates; it is essential, therefore, that all windows created with the application global class be closed before this occurs.
	CS_HREDRAW	Redraws the entire window if the horizontal size changes.
	CS_NOCLOSE	Inhibits the close option on the System menu.
	CS_OWNDC	Gives each window instance its own display context. Note that although the CS_OWNDC style is convenient, it must be used with discretion because each display context occupies approximately 800 bytes of memory.
	CS_PARENTDC	Gives the parent window's display context to the window class.

Field	Description	
	Value	**Meaning**
	CS_SAVEBITS	Saves the portion of the screen image that is obscured by a window; Windows uses the saved bitmap to re-create a screen image when the window is removed. Windows displays the bitmap at its original location and does not send WM_PAINT messages to windows which had been obscured by the window if the memory used by the bitmap has not been discarded and if other screen actions have not invalidated the stored image. An application should set this bit only for small windows that are displayed briefly and then removed before much other screen activity takes place. Setting this bit for a window increases the amount of time required to display the window due to the time required to allocate memory to store the bitmap.
	CS_VREDRAW	Redraws the entire window if the vertical size changes.
lpfnWndProc	Points to the window function.	
cbClsExtra	Specifies the number of bytes to allocate following the window-class structure.	
cbWndExtra	Specifies the number of bytes to allocate following the window instance. If an application is using the **WNDCLASS** structure to register a dialog box created with the **CLASS** directive in the .RC script file, it must set this field to DLGWINDOWEXTRA.	
hInstance	Identifies the class module. The **hInstance** field must be an instance handle and must *not* be NULL.	

Field	Description
hIcon	Identifies the class icon. The **hIcon** field must be a handle to an icon resource. If **hIcon** is NULL, the application must draw an icon whenever the user minimizes the application's window.
hCursor	Identifies the class cursor. The **hCursor** field must be a handle to a cursor resource. If **hCursor** is NULL, the application must explicitly set the cursor shape whenever the mouse moves into the application's window.
hbrBackground	Identifies the class background brush. The **hbrBackground** field can be either a handle to the physical brush that is to be used for painting the background, or it can be a color value. If a color value is given, it must be one of the standard system colors listed below, and the value 1 must be added to the chosen color (for example, COLOR_BACKGROUND + 1 specifies the system background color). If a color value is given, it must be converted to one of the following HBRUSH types: COLOR_ACTIVEBORDER COLOR_ACTIVECAPTION COLOR_APPWORKSPACE COLOR_BACKGROUND COLOR_BTNFACE COLOR_BTNSHADOW COLOR_BTNTEXT COLOR_CAPTIONTEXT COLOR_GRAYTEXT COLOR_HIGHLIGHT COLOR_HIGHLIGHTTEXT COLOR_INACTIVEBORDER COLOR_INACTIVECAPTION COLOR_MENU COLOR_MENUTEXT COLOR_SCROLLBAR COLOR_WINDOW COLOR_WINDOWFRAME COLOR_WINDOWTEXT When **hbrBackground** is NULL, the application must paint its own background whenever it is requested to paint in its client area. The application can determine when the background needs painting by processing the WM_ERASEBKGND message or by testing the **fErase** field of the **PAINTSTRUCT** structure filled by the **BeginPaint** function.

Field	Description
lpszMenuName	Points to a null-terminated character string that specifies the resource name of the class menu (as the name appears in the resource file). If an integer is used to identify the menu, the **MAKEINTRESOURCE** macro can be used. If the **lpszMenuName** field is NULL, windows belonging to this class have no default menu.
lpszClassName	Points to a null-terminated character string that specifies the name of the window class.

Chapter 8 Resource Script Statements

This chapter describes the statements that define resources that the Microsoft Windows Resource Compiler (**RC**) adds to an application's executable file. See *Tools* for information on running the Resource Compiler.

This chapter describes resource script statements in the following categories:

- Single-line statements
- User-defined resources
- RCDATA statement
- STRINGTABLE statement
- ACCELERATORS statement
- Menu statements
- Dialog statements
- Directives

8.1 Single-Line Statements

The single-line statements define resources that are contained in a single file, such as cursors, icons, and fonts. The statements associate the filename of the resource with an identifying name or number. The resource is added to the executable file when the application is created, and can be extracted during execution by referring to the name or number.

The following is the general form for all single-line statements:

nameID resource-type [[*load-option*]] [[*mem-option*]] *filename*

The *nameID* field specifies either a unique name or an integer value identifying the resource. For a font resource, *nameID* must be a number; it cannot be a name.

The *resource-type* field specifies one of the following key words, which identify the type of resource to be loaded:

Key word	Resource Type
CURSOR	Specifies a bitmap that defines the shape of the cursor on the display screen.
ICON	Specifies a bitmap that defines the shape of the icon to be used for a given application.
BITMAP	Specifies a custom bitmap that an application is going to use in its screen display or as an item in a menu.
FONT	Specifies a file that contains a font.

The optional *load-option* field takes a key word that specifies when the resource is to be loaded. The key word must be one of the following:

Option	Description
PRELOAD	Resource is loaded immediately.
LOADONCALL	Resource is loaded when called. This is the default option.

NOTE Icon and cursor resources can contain more than one image. If the resource is marked as **PRELOAD**, Windows loads all images in the resource when the application executes.

The optional *mem-option* field takes the following key word or key words, which specify whether the resource is fixed or moveable and whether it is discardable:

Option	Description
FIXED	Resource remains at a fixed memory location.
MOVEABLE	Resource can be moved if necessary in order to compact memory.
DISCARDABLE	Resource can be discarded if no longer needed.

The default is **MOVEABLE** and **DISCARDABLE** for cursor, icon, and font resources. The default for bitmap resources is **MOVEABLE**.

The *filename* field is an ASCII string that specifies the DOS filename of the file that contains the resource. A full pathname must be given if the file is not in the current working directory.

The following example demonstrates the correct usage for a single-line statement:

```
cursor CURSOR point.cur
cursor CURSOR DISCARDABLE point.cur
10      CURSOR custom.cur

desk    ICON desk.ico
desk    ICON DISCARDABLE desk.ico
11      ICON custom.ico

disk    BITMAP disk.bmp
disk    BITMAP DISCARDABLE disk.bmp
12      BITMAP custom.bmp

5 FONT  CMROMAN.FNT
```

8.2 User-Defined Resources

An application can also define its own resource. The resource can be any data that the application intends to use. A user-defined resource statement has the following form:

nameID typeID [[*load-option*]] [[*mem-option*]] {[[*filename*]] |
[[**BEGIN**
raw-data
END]]}

The *nameID* field specifies either a unique name or an integer value that identifies the resource.

The *typeID* field specifies either a unique name or an integer value that identifies the resource type. If a number is given, it must be greater than 255. The numbers 1 through 255 are reserved for existing and future predefined resource types.

The optional *load-option* field takes a key word that specifies when the resource is to be loaded. The key word must be one of the following:

Option	Description
PRELOAD	Resource is loaded immediately.
LOADONCALL	Resource is loaded when called. This is the default option.

The optional *mem-option* field takes the following key word or key words, which specify whether the resource is fixed or moveable and whether it is discardable:

Option	Description
FIXED	Resource remains at a fixed memory location.
MOVEABLE	Resource can be moved if necessary in order to compact memory. This is the default option.
DISCARDABLE	Resource can be discarded if it is no longer needed.

The optional *filename* field is an ASCII string that specifies the DOS filename of the file that contains the resource. A full pathname must be given if the file is not in the current working directory. Do not use the *filename* field if you supply raw data between the optional **BEGIN** and **END** statements.

The *raw-data* field specifies one or more integers and strings. Integers can be in decimal, octal, or hexadecimal format. Do not use *raw-data* field and the **BEGIN** and **END** statements if you specify a filename.

The following example demonstrates the correct usage for user-defined statements:

```
array   MYRES   data.res
14      300     custom.res
18 MYRES2
BEGIN
  "Here is a data string\0",  /* A string. Note: explicitly
                                 null-terminated */
   1024,                      /* int       */
   0x029a,                    /* hex int   */
   0o733,                     /* octal int */
   "\07"                      /* octal byte */
END
```

8.3 RCDATA Statement

Syntax

nameID **RCDATA** [[*load-option*]] [[*mem-option*]]
BEGIN
raw-data
END

The **RCDATA** statement defines a raw data resource for an application. Raw data resources permit the inclusion of binary data directly in the executable file.

The *nameID* field specifies either a unique name or an integer value that identifies the resource.

The optional *load-option* field takes a key word that specifies when the resource is to be loaded. It must be one of the following:

Option	Description
PRELOAD	Resource is loaded immediately.
LOADONCALL	Resource is loaded when called. This is the default option.

The optional *mem-option* field takes the following key word or key words, which specify whether the resource is fixed or moveable and whether it is discardable:

Option	Description
FIXED	Resource remains at a fixed memory location.
MOVEABLE	Resource can be moved if necessary in order to compact memory.
DISCARDABLE	Resource can be discarded if no longer needed.

The default is **MOVEABLE** and **DISCARDABLE.**

The *raw-data* field specifies one or more integers and strings. Integers can be in decimal, octal, or hexadecimal format.

The following example demonstrates the correct usage for the **RCDATA** statement:

```
resname RCDATA
BEGIN
   "Here is a data string\0", /* A string. Note: explicitly
                                  null-terminated */
   1024,                    /* int       */
   0x029a,                  /* hex int   */
   0o733,                   /* octal int */
   "\07"                    /* octal byte */
END
```

8.4 STRINGTABLE Statement

Syntax

STRINGTABLE [[*load-option*]] [[*mem-option*]]
BEGIN
stringID string
END

The **STRINGTABLE** statement defines one or more string resources for an application. String resources are simply null-terminated ASCII strings that can be loaded when needed from the executable file, using the **LoadString** function.

The optional *load-option* field takes a key word that specifies when the resource is to be loaded. It must be one of the following:

Option	Description
PRELOAD	Resource is loaded immediately.
LOADONCALL	Resource is loaded when called. This is the default option.

The optional *mem-option* field takes the following key word or key words, which specify whether the resource is fixed or moveable and whether or not it is discardable:

Option	Description
FIXED	Resource remains at a fixed memory location.
MOVEABLE	Resource can be moved if necessary in order to compact memory.
DISCARDABLE	Resource can be discarded if no longer needed.

The default is **MOVEABLE** and **DISCARDABLE.**

The *stringID* field specifies an integer value that identifies the resource.

The *string* field specifies one or more ASCII strings, enclosed in double quotation marks. The string must be no longer than 255 characters and must occupy a single line in the source file. To add a carriage return to the string, use this character sequence: \012. For example, "Line one\012Line two" would define a string that would be displayed as follows:

Line one
Line two

Grouping strings in separate segments allows all related strings to be read in at one time and discarded together. When possible, an application should make the table moveable and discardable. The Resource Compiler allocates 16 strings per segment and uses the identifier value to determine which segment is to contain the string. Strings with the same upper 12 bits in their identifiers are placed in the same segment.

The following example demonstrates the correct usage of the STRINGTABLE statement:

```
#define IDS_HELLO    1
#define IDS_GOODBYE  2

STRINGTABLE
BEGIN
```

```
    IDS_HELLO,   "Hello"
    IDS_GOODBYE, "Goodbye"
END
```

8.5 ACCELERATORS Statement

Syntax

acctablename **ACCELERATORS**
BEGIN
event, *idvalue*, [[*type*]] [[**NOINVERT**]] [[**ALT**]] [[**SHIFT**]] [[**CONTROL**]]
.
.
.
END

The **ACCELERATORS** statement defines one or more accelerators for an application. An accelerator is a key stroke defined by the application to give the user a quick way to perform a task. The **TranslateAccelerator** function is used to translate accelerator messages from the application queue into WM_COMMAND or WM_SYSCOMMAND messages.

The *acctablename* field specifies either a unique name or an integer value that identifies the resource.

The *event* field specifies the key stroke to be used as an accelerator. It can be any one of the following:

Character	Description
"char"	A single ASCII character enclosed in double quotes. The character can be preceded by a caret (^), meaning that the character is a control character.
ASCII character	An integer value representing an ASCII character. The *type* field must be **ASCII**.
Virtual key character	An integer value representing a virtual key. The virtual key for alphanumeric keys can be specified by placing the uppercase letter or number in double quotation marks (for example, "9" or "C"). The *type* field must be **VIRTKEY**.

The *idvalue* field specifies an integer value that identifies the accelerator.

The *type* field is required only when *event* is an ASCII character or a virtual key character. The *type* field specifies either **ASCII** or **VIRTKEY**; the integer value of *event* is interpreted accordingly. When **VIRTKEY** is specified and the *event* field contains a string, the *event* field must be uppercase.

The **NOINVERT** option, if given, means that no top-level menu item is highlighted when the accelerator is used. This is useful when defining accelerators for actions such as scrolling that do not correspond to a menu item. If **NOINVERT** is omitted, a top-level menu item will be highlighted (if possible) when the accelerator is used.

The **ALT** option, if given, causes the accelerator to be activated only if the ALT key is down.

The **SHIFT** option, if given, causes the accelerator to be activated only if the SHIFT key is down.

The **CONTROL** option, if given, defines the character as a control character (the accelerator is only activated if the CONTROL key is down). This has the same effect as using a caret (^) before the accelerator character in the *event* field.

The **ALT**, **SHIFT**, and **CONTROL** options apply only to virtual keys.

The following example demonstrates the correct usage of accelerator keys:

```
1 ACCELERATORS
BEGIN
  "^C",  IDDCLEAR          ; control C
  "K",   IDDCLEAR          ; shift K
  "k",   IDDELLIPSE, ALT   ; alt K
  98,    IDDRECT, ASCII    ; b
  66,    IDDSTAR, ASCII    ; B (shift b)
  "g",   IDDRECT           ; g
  "G",   IDDSTAR           ; G (shift G)
  VK_F1, IDDCLEAR, VIRTKEY                  ; F1
  VK_F1, IDDSTAR,  CONTROL, VIRTKEY         ; control F1
  VK_F1, IDDELLIPSE,  SHIFT, VIRTKEY        ; shift F1
  VK_F1, IDDRECT,  ALT, VIRTKEY             ; alt F1
  VK_F2, IDDCLEAR, ALT, SHIFT,     VIRTKEY  ; alt shift F2
  VK_F2, IDDSTAR,  CONTROL, SHIFT, VIRTKEY  ; ctrl shift F2
  VK_F2, IDDRECT,  ALT, CONTROL,   VIRTKEY  ; alt control F2
END
```

8.6 MENU Statement

Syntax

menuID **MENU** [[*load-option*]] [[*mem-option*]]
BEGIN
item-definitions
END

The **MENU** statement defines the contents of a menu resource. A menu resource is a collection of information that defines the appearance and function of an application menu. A menu is a special input tool that lets a user select commands from a list of command names.

The *menuID* field specifies a name or number used to identify the menu resource.

The optional *load-option* field takes a key word that specifies when the resource is to be loaded. It must be one of the following:

Option	Description
PRELOAD	Resource is loaded immediately.
LOADONCALL	Resource is loaded when called. This is the default option.

The optional *mem-option* field takes the following key word or key words, which specify whether the resource is fixed or moveable and whether it is discardable:

Option	Description
FIXED	Resource remains at a fixed memory location.
MOVEABLE	Resource can be moved if necessary in order to compact memory.
DISCARDABLE	Resource can be discarded if no longer needed.

The default is **MOVEABLE** and **DISCARDABLE.**

The *item-definition* field specifies special resource statements that define the items in the menu. These statements are defined in the following sections.

The following is an example of a complete **MENU** statement:

```
sample MENU
BEGIN
     MENUITEM "&Soup", 100
     MENUITEM "S&alad", 101
     POPUP "&Entree"
     BEGIN
          MENUITEM "&Fish", 200
          MENUITEM "&Chicken", 201, CHECKED
          POPUP "&Beef"
          BEGIN
               MENUITEM "&Steak", 301
               MENUITEM "&Prime Rib", 302
          END
     END
     MENUITEM "&Dessert", 103
END
```

8.6.1 Item-Definition Statements

The **MENUITEM** and **POPUP** statements are used in the *item-definition* section of a **MENU** statement to define the names and attributes of the actual menu items. Any number of statements can be given; each defines a unique item. The order of the statements defines the order of the menu items.

The **MENUITEM** and **POPUP** statements can be used only within an *item-definition* section of a **MENU** statement.

MENUITEM Statement

Syntax

MENUITEM *text*, *result*, [[*optionlist*]]

This optional statement defines a menu item.

The *text* field takes an ASCII string, enclosed in double quotation marks, that specifies the name of the menu item.

The string can contain the escape characters **\t** and **\a.** The **\t** character inserts a tab in the string and is used to align text in columns. Tab characters should be used only in pop-up menus, not in menu bars. (See the following section for information on pop-up menus.) The **\a** character aligns all text that follows it flush right to the menu bar or pop-up menu.

To insert a double quotation mark (") in the string, use two double quotation marks ("").

To add a mnemonic to the text string, place the ampersand (&) ahead of the letter that will be the mnemonic. This will cause the letter to appear underlined in the control and to function as the mnemonic. To use the ampersand as a character in a string, insert two ampersands (&&).

The *result* field takes an integer value that specifies the result generated when the user selects the menu item. Menu-item results are always integers; when the user clicks the menu-item name, the result is sent to the window that owns the menu.

The optional *optionlist* field takes one or more predefined menu options, separated by commas or spaces, that specify the appearance of the menu item. The menu options are as follows:

Option	Description
CHECKED	Item has a checkmark next to it.
GRAYED	Item name is initially inactive and appears on the menu in gray or a lightened shade of the menu-text color.

Option	Description
HELP	Item has a vertical separator to its left.
INACTIVE	Item name is displayed, but it cannot be selected.
MENUBARBREAK	Same as MF_MENUBREAK except that for pop-up menus, it separates the new column from the old column with a vertical line.
MENUBREAK	Places the menu item on a new line for static menu-bar items. For pop-up menus, places the menu item in a new column, with no dividing line between the columns.

The **INACTIVE** and **GRAYED** options cannot be used together.

The following example demonstrates the correct usage of the **MENUITEM** statement:

```
MENUITEM  "&Alpha", 1, CHECKED, GRAYED
MENUITEM  "&Beta", 2
```

POPUP Statement

Syntax

POPUP *text*, [[*optionlist*]]
BEGIN
item-definitions
END

This statement marks the beginning of the definition of a pop-up menu. A pop-up menu (which is also known as a drop-down menu) is a special menu item that displays a sublist of menu items when it is selected.

The *text* field takes an ASCII string, enclosed in double quotation marks, that specifies the name of the pop-up menu.

The optional *optionlist* field takes one or more predefined menu options that specify the appearance of the menu item. The menu options are as follows:

Option	Description
CHECKED	Item has a checkmark next to it. This option is not valid for a top-level pop-up menu.
GRAYED	Item name is initially inactive and appears on the menu in gray or a lightened shade of the menu-text color.

Option	Description
INACTIVE	Item name is displayed, but it cannot be selected.
MENUBARBREAK	Same as MF_MENUBREAK except that for pop-up menus, it separates the new column from the old column with a vertical line.
MENUBREAK	Places the menu item on a new line for static menu-bar items. For pop-up menus, places the menu item in a new column, with no dividing line between the columns.

The options can be combined using the bitwise OR operator. The **INACTIVE** and **GRAYED** options cannot be used together.

The *item-definitions* field can specify any number of **MENUITEM** or **POPUP** statements. As a result, any pop-up menu item can display another pop-up menu.

The following example demonstrates the correct usage of the **POPUP** statement:

```
chem MENU
BEGIN

POPUP "&Elements"
BEGIN
     MENUITEM "&Oxygen", 200
     MENUITEM "&Carbon", 201, CHECKED
     MENUITEM "&Hydrogen", 202
     MENUITEM "&Sulfur", 203
     MENUITEM "Ch&lorine", 204
END

POPUP "&Compounds"
BEGIN
     POPUP "&Sugars"
     BEGIN
        MENUITEM "&Glucose", 301
        MENUITEM "&Sucrose", 302, CHECKED
        MENUITEM "&Lactose", 303, MENUBREAK
        MENUITEM "&Fructose", 304
     END

     POPUP "&Acids"
     BEGIN
          "&Hydrochloric", 401
          "&Sulfuric", 402
     END

END

END
```

MENUITEM SEPARATOR Statement

Syntax

MENUITEM SEPARATOR

This special form of the **MENUITEM** statement creates an inactive menu item that serves as a dividing bar between two active menu items in a pop-up menu.

The following demonstrates the correct usage of the **MENUITEM SEPARATOR** statement:

```
MENUITEM "&Roman", 206
MENUITEM SEPARATOR
MENUITEM "&20 Point", 301
```

8.7 DIALOG Statement

The **DIALOG** statement defines a template that can be used by an application to create dialog boxes.

Syntax

nameID **DIALOG** [[*load-option*]] [[*mem-option*]] *x*, *y*, *width*, *height*
[[*option-statements*]]
BEGIN
control-statements
END

This statement marks the beginning of a **DIALOG** template. It defines the name of the dialog box, the memory and load options, the box's starting location on the display screen, and the box's width and height.

The *nameID* field specifies either a unique name or an integer value that identifies the resource.

The optional *load-option* field takes a key word that specifies when the resource is to be loaded. It must be one of the following:

Option	Description
PRELOAD	Resource is loaded immediately.
LOADONCALL	Resource is loaded when called. This is the default option.

The optional *mem-option* field takes the following key word or key words, which specify whether the resource is fixed or moveable and whether it is discardable:

Option	Description
FIXED	Resource remains at a fixed memory location.
MOVEABLE	Resource can be moved if necessary in order to compact memory. This is the default option.
DISCARDABLE	Resource can be discarded if no longer needed.

The *x* and *y* fields take integer values that specify the *x* and *y* coordinates of the upper-left corner of the dialog box. The horizontal units are 1/4 of the dialog base width unit; the vertical units are 1/8 of the dialog base height unit. The current dialog base units are computed from the height and width of the current system font. The **GetDialogBaseUnits** function returns the dialog base units in pixels. The exact meaning of the coordinates depends on the style defined by the **STYLE** option statement. For child-style dialog boxes, the coordinates are relative to the origin of the parent window, unless the dialog box has the style DS_ABSALIGN; in that case, the coordinates are relative to the origin of the display screen.

The *width* and *height* fields take integer values that specify the width and height of the box. The width units are 1/4 of the dialog base width unit; the height units are 1/8 of the dialog base height unit.

The option and control statements are described in the following sections.

The following demonstrates the correct usage of the **DIALOG** statement:

```
#include "WINDOWS.H"

errmess DIALOG  10, 10, 300, 110
STYLE WS_POPUP|WS_BORDER
CAPTION "Error!"
BEGIN
    CTEXT "Select One:", 1, 10, 10, 280, 12
    RADIOBUTTON "&Retry", 2, 75, 30, 60, 12
    RADIOBUTTON "&Abort", 3, 75, 50, 60, 12
    RADIOBUTTON "&Ignore", 4, 75, 80, 60, 12
END
```

Comments

Do not use the WS_CHILD style with a modal dialog box. The **DialogBox** function always disables the parent/owner of the newly-created dialog box. When a parent window is disabled, its child windows are implicitly disabled. Since the parent window of the child-style dialog box is disabled, the child-style dialog box is too.

If a dialog box has the DS_ABSALIGN style, the dialog coordinates for its upper-left corner are relative to the screen origin instead of to the upper-left corner of the parent window. You would typically use this style when you wanted the dialog box to start in a specific part of the display no matter where the parent window may be on the screen.

The name **DIALOG** can also be used as the class-name parameter to the **CreateWindow** function in order to create a window with dialog-box attributes.

8.7.1 Dialog Option Statements

The dialog option statements, given in the *option-statements* section of the **DIALOG** statement, define special attributes of the dialog box, such as its style, caption, and menu. The option statements are optional. If the application does not supply a particular option statement, the dialog box is given default attributes for that option. Dialog option statements include the following:

- **STYLE**
- **CAPTION**
- **MENU**
- **CLASS**
- **FONT**

The option statements are discussed individually in the following sections.

STYLE Statement

Syntax

STYLE *style*

This optional statement defines the window style of the dialog box. The window style specifies whether the box is a pop-up or a child window. The default style has the following attributes:

WS_POPUP
WS_BORDER
WS_SYSMENU

The *style* field takes an integer value or predefined name that specifies the window style. It can be any of the window styles defined in Table 8.1, "Window Styles."

Comments

If the predefined names are used, the **#include** directive must be used so that the WINDOWS.H file will be included in the resource script.

Table 8.1 Window Styles

Style	Meaning
DS_LOCALEDIT	Specifies that edit controls in the dialog box will use memory in the application's data segment. By default, all edit controls in dialog boxes use memory outside the application's data segment. This feature can be suppressed by adding the DS_LOCALEDIT flag to the STYLE command for the dialog box. If this flag is not used, EM_GETHANDLE and EM_SETHANDLE messages must not be used since the storage for the control is not in the application's data segment. This feature does not affect edit controls created outside of dialog boxes.
DS_MODALFRAME	Creates a dialog box with a modal dialog-box frame that can be combined with a title bar and system menu by specifying the WS_CAPTION and WS_SYSMENU styles.
DS_NOIDLEMSG	Suppresses WM_ENTERIDLE messages that Windows would otherwise send to the owner of the dialog box while the dialog box is displayed.
DS_SYSMODAL	Creates a system-modal dialog box.
WS_BORDER	Creates a window that has a border.
WS_CAPTION	Creates a window that has a title bar (implies WS_BORDER).
WS_CHILD	Creates a child window. It cannot be used with WS_POPUP.
WS_CHILDWINDOW	Creates a child window that has the style WS_CHILD.
WS_CLIPCHILDREN	Excludes the area occupied by child windows when drawing within the parent window. Used when creating the parent window.
WS_CLIPSIBLINGS	Clips child windows relative to each other; that is, when a particular child window receives a WP_PAINT message, this style clips all other top-level child windows out of the region of the child window to be updated. (If WS_CLIPSIBLINGS is not given and child windows overlap, it is possible, when drawing in the client area of a child window, to draw in the client area of a neighboring child window.) For use with WS_CHILD only.
WS_DISABLED	Creates a window that is initially disabled.
WS_DLGFRAME	Creates a window with a modal dialog-box frame but no title.

Table 8.1 Window Styles *(continued)*

Style	Meaning
WS_GROUP	Specifies the first control of a group of controls in which the user can move from one control to the next by using the arrow keys. All controls defined with the WS_GROUP style after the first control belong to the same group. The next control with the WS_GROUP style ends the style group and starts the next group (i.e., one group ends where the next begins). This style is valid only for controls.
WS_HSCROLL	Creates a window that has a horizontal scroll bar.
WS_ICONIC	Creates a window that is initially iconic. For use with WS_OVERLAPPED only.
WS_MAXIMIZE	Creates a window of maximum size.
WS_MAXIMIZEBOX	Creates a window that has a Maximize box.
WS_MINIMIZE	Creates a window of minimum size.
WS_MINIMIZEBOX	Creates a window that has a Minimize box.
WS_OVERLAPPED	Creates an overlapped window. An overlapped window has a caption and a border.
WS_OVERLAPPEDWINDOW	Creates an overlapped window having the WS_OVERLAPPED, WS_CAPTION, WS_SYSMENU, WS_THICKFRAME, WS_MINIMIZEBOX, and WS_MAXIMIZEBOX styles.
WS_POPUP	Creates a pop-up window. It cannot be used with WS_CHILD.
WS_POPUPWINDOW	Creates a pop-up window that has the styles WS_POPUP, WS_BORDER, and WS_SYSMENU. The WS_CAPTION style must be combined with the WS_POPUPWINDOW style to make the system menu visible.
WS_SIZEBOX	Creates a window that has a size box. Used only for windows with a title bar or with vertical and horizontal scroll bars.
WS_SYSMENU	Creates a window that has a System-menu box in its title bar. Used only for windows with title bars. If used with a child window, this style creates a Close box instead of a System-menu box.
WS_TABSTOP	Specifies one of any number of controls through which the user can move by using the TAB key. The TAB key moves the user to the next control specified by the WS_TABSTOP style. This style is valid only for controls.

Table 8.1 Window Styles *(continued)*

Style	Meaning
WS_THICKFRAME	Creates a window with a thick frame that can be used to size the window.
WS_VISIBLE	Creates a window that is initially visible. This applies to overlapping and pop-up windows. For overlapping windows, the *y* parameter is used as a **ShowWindow** function parameter.
WS_VSCROLL	Creates a window that has a vertical scroll bar.

CAPTION Statement

Syntax

CAPTION *captiontext*

This optional statement defines the dialog box's title. The title appears in the box's caption bar (if it has one).

The default caption is empty.

The *captiontext* field specifies an ASCII character string enclosed in double quotation marks.

The following example demonstrates the correct usage of the **CAPTION** statement:

```
CAPTION "Error!"
```

MENU Statement

Syntax

MENU *menuname*

This optional statement defines the dialog box's menu. If no statement is given, the dialog box has no menu.

The *menuname* field specifies the resource name or number of the menu to be used.

The following example demonstrates the correct usage of the **MENU** statement:

```
MENU errmenu
```

CLASS Statement

Syntax

CLASS *class*

This optional statement defines the class of the dialog box. If no statement is given, the Windows standard dialog class will be used as the default.

The *class* field specifies an integer or a string, enclosed in double quotation marks, that identifies the class of the dialog box. If the window procedure for the class does not process a message sent to it, it must call the **DefDlgProc** function to ensure that all messages are handled properly for the dialog box. A private class can use **DefDlgProc** as the default window procedure. The class must be registered with the **cbWndExtra** field of the **WNDCLASS** data structure set to DLGWINDOWEXTRA.

The following example demonstrates the correct usage of the **CLASS** statement:

```
CLASS "myclass"
```

Comments

The **CLASS** statement should be used with special cases, since it overrides the normal processing of a dialog box. The **CLASS** statement converts a dialog box to a window of the specified class; depending on the class, this could give undesirable results. Do not use the predefined control class names with this statement.

FONT Statement

Syntax

FONT *pointsize*, *typeface*

This optional statement defines the font with which Windows will draw text in the dialog box. The font must have been previously loaded, either from WIN.INI or by calling **LoadFont.**

The *pointsize* field is an integer that specifies the size in points of the font.

The *typeface* field specifies an ASCII character string enclosed in double quotation marks that specifies the name of the typeface. This name must be identical to the name defined in the [fonts] section of WIN.INI.

The following example demonstrates the correct usage of the **FONT** statement:

```
FONT 12, "Helv"
```

8.7.2 Dialog Control Statements

The dialog control statements, given in the *control-statements* section of the **DIALOG** statement, define the attributes of the control windows that appear in the dialog box. A dialog box is empty unless one or more control statements are given. Control statements include the following:

- **LTEXT**
- **RTEXT**
- **CTEXT**
- **CHECKBOX**
- **PUSHBUTTON**
- **LISTBOX**
- **GROUPBOX**
- **DEFPUSHBUTTON**
- **RADIOBUTTON**
- **EDITTEXT**
- **COMBOBOX**
- **ICON**
- **SCROLLBAR**
- **CONTROL**

The control statements are discussed individually in the following sections. For more information on control classes and styles, see Tables 8.2, "Control Classes," and 8.3, "Control Styles."

LTEXT Statement

Syntax

LTEXT *text*, *id*, *x*, *y*, *width*, *height*, [[*style*]]

This statement defines a flush-left text control. It creates a simple rectangle that displays the given text flush-left in the rectangle. The text is formatted before it is displayed. Words that would extend past the end of a line are automatically wrapped to the beginning of the next line.

The *text* field takes an ASCII string that specifies the text to be displayed. The string must be enclosed in double quotation marks. To add a mnemonic to the

text string, place the ampersand (&) ahead of the letter that will be the mnemonic. To use the ampersand as a character in a string, insert two ampersands (&&).

The *id* field takes a unique integer value that identifies the control.

The *x* and *y* fields take integer values that specify the *x* and *y* coordinates of the upper-left corner of the control. The horizontal units are ¼ of the dialog base width unit; the vertical units are ⅛ of the dialog base height unit. The current dialog base units are computed from the height and width of the current system font. The **GetDialogBaseUnits** function returns the dialog base units in pixels. The coordinates are relative to the origin of the dialog box.

The *width* and *height* fields take integer values that specify the width and height of the control. The width units are ¼ of the dialog base width unit; the height units are ⅛ of the dialog base height unit.

The optional *style* field can contain any combination (or none) of the following styles:

- WS_TABSTOP
- WS_GROUP

These styles are described in Table 8.1, "Window Styles." Styles can be combined using the bitwise OR operator.

Comments

The *x*, *y*, *width*, and *height* fields can use the addition operator (+) for relative positioning. For example, "15 + 6" can be used for the *x* field.

The default style for **LTEXT** is SS_LEFT and WS_GROUP.

The following example demonstrates the correct usage of the **LTEXT** statement:

```
LTEXT "Enter Name:", 3, 10, 10, 40, 10
```

RTEXT Statement

Syntax

RTEXT *text*, *id*, *x*, *y*, *width*, *height*, [[*style*]]

This statement defines a flush-right text control. It creates a simple rectangle that displays the given text flush-right in the rectangle. The text is formatted before it is displayed. Words that would extend past the end of a line are automatically wrapped to the beginning of the next line.

The *text* field takes an ASCII string that specifies the text to be displayed. The string must be enclosed in double quotation marks. To add a mnemonic to the

text string, place the ampersand (&) ahead of the letter that will be the mnemonic. To use the ampersand as a character in a string, insert two ampersands (&&).

The *id* field takes a unique integer value that identifies the control.

The *x* and *y* fields take integer values that specify the *x* and *y* coordinates of the upper-left corner of the control. The horizontal units are 1/4 of the dialog base width unit; the vertical units are 1/8 of the dialog base height unit. The current dialog base units are computed from the height and width of the current system font. The **GetDialogBaseUnits** function returns the dialog base units in pixels. The coordinates are relative to the origin of the dialog box.

The *width* and *height* fields take integer values that specify the width and height of the control. The width units are 1/4 of the dialog base width unit; the height units are 1/8 of the dialog base height unit.

The optional *style* field can contain any combination (or none) of the following styles:

- WS_TABSTOP
- WS_GROUP

These styles are described in Table 8.1, "Window Styles." Styles can be combined using the bitwise OR operator.

Comments

The *x*, *y*, *width*, and *height* fields can use the addition operator (+) for relative positioning. For example, "15 + 6" can be used for the *x* field.

The default style for **RTEXT** is SS_RIGHT and WS_GROUP.

The following example demonstrates the correct usage of the **RTEXT** statement:

```
RTEXT "Number of Messages", 4, 30, 50, 100, 10
```

CTEXT Statement

Syntax

CTEXT *text*, *id*, *x*, *y*, *width*, *height*, [[*style*]]

This statement defines a centered text control. It creates a simple rectangle that displays the given text centered in the rectangle. The text is formatted before it is displayed. Words that would extend past the end of a line are automatically wrapped to the beginning of the next line.

The *text* field takes an ASCII string that specifies the text to be displayed. The string must be enclosed in double quotation marks. To add a mnemonic to the

text string, place the ampersand (&) ahead of the letter that will be the mnemonic. To use the ampersand as a character in a string, insert two ampersands (&&).

The *id* field takes a unique integer value that identifies the control.

The *x* and *y* fields take integer values that specify the *x* and *y* coordinates of the upper-left corner of the control. The horizontal units are ¼ of the dialog base width unit; the vertical units are ⅛ of the dialog base height unit. The current dialog base units are computed from the height and width of the current system font. The **GetDialogBaseUnits** function returns the dialog base units in pixels. The coordinates are relative to the origin of the dialog box.

The *width* and *height* fields take integer values that specify the width and height of the control. The width units are ¼ of the dialog base width unit; the height units are ⅛ of the dialog base height unit.

The optional *style* field can contain any combination (or none) of the following styles:

- WS_TABSTOP
- WS_GROUP

These styles are described in Table 8.1, "Window Styles." Styles can be combined using the bitwise OR operator.

Comments

The *x*, *y*, *width*, and *height* fields can use the addition operator (+) for relative positioning. For example, "15 + 6" can be used for the *x* field.

The default style for **CTEXT** is SS_CENTER and WS_GROUP.

The following example demonstrates the correct usage of the **CTEXT** statement:

```
CTEXT "Title", 3, 10, 50, 40, 10
```

CHECKBOX Statement

Syntax

CHECKBOX *text*, *id*, *x*, *y*, *width*, *height*, [[*style*]]

This statement defines a check-box control belonging to the BUTTON class. It creates a small rectangle (check box) that is highlighted when clicked. The given text is displayed just to the right of the check box. The control highlights the rectangle when the user clicks the mouse in it, and removes the highlight on the next click.

The *text* field takes an ASCII string that specifies the text to be displayed. The string must be enclosed in double quotation marks. To add a mnemonic to the text string, place the ampersand (&) ahead of the letter that will be the mnemonic. To use the ampersand as a character in a string, insert two ampersands (&&).

The *id* field takes a unique integer value that identifies the control.

The *x* and *y* fields take integer values that specify the *x* and *y* coordinates of the upper-left corner of the control. The horizontal units are 1/4 of the dialog base width unit; the vertical units are 1/8 of the dialog base height unit. The current dialog base units are computed from the height and width of the current system font. The **GetDialogBaseUnits** function returns the dialog base units in pixels. The coordinates are relative to the origin of the dialog box.

The *width* and *height* fields take integer values that specify the width and height of the control. The width units are 1/4 of the dialog base width unit; the height units are 1/8 of the dialog base height unit.

The optional *style* field can contain any combination (or none) of the following styles:

- WS_TABSTOP
- WS_GROUP

These styles are described in Table 8.1, "Window Styles."

In addition to these styles, the *style* field may contain any combination (or none) of the BUTTON-class styles described in Table 8.3, "Control Styles." Styles can be combined using the bitwise OR operator.

Comments

The *x*, *y*, *width*, and *height* fields can use the addition operator (+) for relative positioning. For example, "15 + 6" can be used for the *x* field.

The default style for **CHECKBOX** is BS_CHECKBOX and WS_TABSTOP.

The following example demonstrates the correct usage of the **CHECKBOX** statement:

```
CHECKBOX "Arabic", 3, 10, 10, 40, 10
```

PUSHBUTTON Statement

Syntax

PUSHBUTTON *text, id, x, y, width, height,* [[*style*]]

This statement defines a push-button control belonging to the BUTTON class. It creates a rectangle containing the given text. The control sends a message to its parent whenever the user clicks the mouse inside the rectangle.

The *text* field takes an ASCII string that specifies the text to be displayed. The string must be enclosed in double quotation marks. To add a mnemonic to the text string, place the ampersand (&) ahead of the letter that will be the mnemonic. To use the ampersand as a character in a string, insert two ampersands (&&).

The *id* field takes a unique integer value that identifies the control.

The *x* and *y* fields take integer values that specify the *x* and *y* coordinates of the upper-left corner of the control. The horizontal units are ¼ of the dialog base width unit; the vertical units are ⅛ of the dialog base height unit. The current dialog base units are computed from the height and width of the current system font. The **GetDialogBaseUnits** function returns the dialog base units in pixels. The coordinates are relative to the origin of the dialog box.

The *width* and *height* fields take integer values that specify the width and height of the control. The width units are ¼ of the dialog base width unit; the height units are ⅛ of the dialog base height unit.

The optional *style* field can contain any combination (or none) of the following styles:

- WS_TABSTOP
- WS_DISABLED
- WS_GROUP

These styles are described in Table 8.1, "Window Styles."

In addition to these styles, the *style* field may contain any combination (or none) of the BUTTON-class styles described in Table 8.3, "Control Styles." Styles can be combined using the bitwise OR operator.

Comments

The *x*, *y*, *width*, and *height* fields can use the addition operator (+) for relative positioning. For example, "15 + 6" can be used for the *x* field.

The default style for **PUSHBUTTON** is BS_PUSHBUTTON and WS_TABSTOP.

The following example demonstrates the correct usage of the **PUSHBUTTON** statement:

```
PUSHBUTTON "ON", 7, 10, 10, 20, 10
```

LISTBOX Statement

Syntax

LISTBOX *id, x, y, width, height,* [[*style*]]

This statement defines a list box belonging to the LISTBOX class. It creates a rectangle that contains a list of strings (such as filenames) from which the user can make selections.

The *id* field takes a unique integer value that identifies the control.

The *x* and *y* fields take integer values that specify the *x* and *y* coordinates of the upper-left corner of the control. The horizontal units are ¼ of the dialog base width unit; the vertical units are ⅛ of the dialog base height unit. The current dialog base units are computed from the height and width of the current system font. The **GetDialogBaseUnits** function returns the dialog base units in pixels. The coordinates are relative to the origin of the dialog box.

The *width* and *height* fields take integer values that specify the width and height of the control. The width units are ¼ of the dialog base width unit; the height units are ⅛ of the dialog base height unit.

The optional *style* field can contain any combination (or none) of the following styles:

- WS_BORDER
- WS_VSCROLL

These styles are described in Table 8.1, "Window Styles."

In addition to these styles, the *style* field may contain any combination (or none) of the LISTBOX-class styles described in Table 8.3, "Control Styles." Styles can be combined using the bitwise OR operator.

Comments

The *x*, *y*, *width*, and *height* fields can use the addition operator (+) for relative positioning. For example, "15 + 6" can be used for the *x* field.

The default style for **LISTBOX** is LBS_NOTIFY, WS_VSCROLL, and WS_BORDER.

For information on the recommended keys for use in list-box controls, see the *System Application Architecture, Common User Access: Advanced Interface Design Guide.*

The following example demonstrates the correct usage of the **LISTBOX** statement:

```
LISTBOX 666, 10, 10, 50, 54
```

GROUPBOX Statement

Syntax

GROUPBOX *text, id, x, y, width, height,* [[*style*]]

This statement defines a group box belonging to the BUTTON class. It creates a rectangle that groups other controls together. The controls are grouped by drawing a border around them and displaying the given text in the upper-left corner.

The *text* field takes an ASCII string that specifies the text to be displayed. The string must be enclosed in double quotation marks. To add a mnemonic to the text string, place the ampersand (&) ahead of the letter that will be the mnemonic. Selecting the mnemonic moves the input focus to the next control in the group, in the order set in the resource file. To use the ampersand as a character in a string, insert two ampersands (&&).

The *id* field takes a unique integer value that identifies the control.

The *x* and *y* fields take integer values that specify the *x* and *y* coordinates of the upper-left corner of the control. The horizontal units are ¼ of the dialog base width unit; the vertical units are ⅛ of the dialog base height unit. The current dialog base units are computed from the height and width of the current system font. The **GetDialogBaseUnits** function returns the dialog base units in pixels. The coordinates are relative to the origin of the dialog box.

The *width* and *height* fields take integer values that specify the width and height of the control. The width units are ¼ of the dialog base width unit; the height units are ⅛ of the dialog base height unit.

The optional *style* field can contain any combination (or none) of the following styles:

- WS_TABSTOP
- WS_DISABLED

These styles are described in Table 8.1, "Window Styles."

In addition to these styles, the *style* field may contain any combination (or none) of the BUTTON-class styles described in Table 8.3, "Control Styles." Styles can be combined using the bitwise OR operator.

Comments

The *x*, *y*, *width*, and *height* fields can use the addition operator (+) for relative positioning. For example, "15 + 6" can be used for the *x* field.

The default style for **GROUPBOX** is BS_GROUPBOX and WS_TABSTOP.

The following example demonstrates the correct usage of the **GROUPBOX** statement:

```
GROUPBOX "Output", 42, 10, 10, 30, 50
```

DEFPUSHBUTTON Statement

Syntax

DEFPUSHBUTTON *text*, *id*, *x*, *y*, *width*, *height*, [[*style*]]

This statement defines a default push-button control that belongs to the BUTTON class. It creates a small rectangle with a bold outline that represents the default response for the user. The given text is displayed inside the button. The control highlights the button in the usual way when the user clicks the mouse in it and sends a message to its parent window.

The *text* field takes an ASCII string that specifies the text to be displayed. The string must be enclosed in double quotation marks. To add a mnemonic to the text string, place the ampersand (&) ahead of the letter that will be the mnemonic. To use the ampersand as a character in a string, insert two ampersands (&&).

The *id* field takes a unique integer value that identifies the control.

The *x* and *y* fields take integer values that specify the *x* and *y* coordinates of the upper-left corner of the control. The horizontal units are ¼ of the dialog base width unit; the vertical units are ⅛ of the dialog base height unit. The current dialog base units are computed from the height and width of the current system font. The **GetDialogBaseUnits** function returns the dialog base units in pixels. The coordinates are relative to the origin of the dialog box.

The *width* and *height* fields take integer values that specify the width and height of the control. The width units are ¼ of the dialog base width unit; the height units are ⅛ of the dialog base height unit.

The optional *style* field can contain any combination (or none) of the following styles:

- WS_TABSTOP
- WS_GROUP
- WS_DISABLED

These styles are described in Table 8.1, "Window Styles."

In addition to these styles, the *style* field may contain any combination (or none) of the BUTTON-class styles described in Table 8.3, "Control Styles." Styles can be combined using the bitwise OR operator.

Comments

The *x*, *y*, *width*, and *height* fields can use the addition operator (+) for relative positioning. For example, "15 + 6" can be used for the *x* field.

The default style for **DEFPUSHBUTTON** is BS_DEFPUSHBUTTON and WS_TABSTOP.

The following example demonstrates the correct usage of the **DEFPUSHBUTTON** statement:

```
DEFPUSHBUTTON "ON", 7, 10, 10, 20, 10
```

RADIOBUTTON Statement

Syntax

RADIOBUTTON *text*, *id*, *x*, *y*, *width*, *height*, [[*style*]]

This statement defines a radio-button control belonging to the BUTTON class. It creates a small circle that has the given text displayed just to its right. The control highlights the button when the user clicks the mouse in it and sends a message to its parent window. The control removes the highlight and sends a message on the next click.

The *text* field takes an ASCII string that specifies the text to be displayed. The string must be enclosed in double quotation marks. To add a mnemonic to the text string, place the ampersand (&) ahead of the letter that will be the mnemonic. To use the ampersand as a character in a string, insert two ampersands (&&).

The *id* field takes a unique integer value that identifies the control.

The *x* and *y* fields take integer values that specify the *x* and *y* coordinates of the upper-left corner of the control. The horizontal units are 1/4 of the dialog base width unit; the vertical units are 1/8 of the dialog base height unit. The current dialog base units are computed from the height and width of the current system font. The **GetDialogBaseUnits** function returns the dialog base units in pixels. The coordinates are relative to the origin of the dialog box.

The *width* and *height* fields take integer values that specify the width and height of the control. The width units are 1/4 of the dialog base width unit; the height units are 1/8 of the dialog base height unit.

The optional *style* field can contain any combination (or none) of the following styles:

- WS_TABSTOP
- WS_GROUP
- WS_DISABLED

These styles are described in Table 8.1, "Window Styles."

In addition to these styles, the *style* field may contain any combination (or none) of the BUTTON-class styles described in Table 8.3, "Control Styles." Styles can be combined using the bitwise OR operator.

Comments

The *x*, *y*, *width*, and *height* fields can use the addition operator (+) for relative positioning. For example, "15 + 6" can be used for the *x* field.

The default style for **RADIOBUTTON** is BS_RADIOBUTTON and WS_TAB-STOP.

The following example demonstrates the correct usage of the **RADIOBUTTON** statement:

```
RADIOBUTTON "AM 101", 10, 10, 10, 40, 10
```

EDITTEXT Statement

Syntax

EDITTEXT *id*, *x*, *y*, *width*, *height*, [[*style*]]

This statement defines an EDIT control belonging to the EDIT class. It creates a rectangular region in which the user can enter and edit text. The control displays a cursor when the user clicks the mouse in it. The user can then use the keyboard to enter text or edit the existing text. Editing keys include the BACKSPACE and DELETE keys. The user can also use the mouse to select characters to be deleted, or to select the place to insert new characters.

The *id* field takes a unique integer value that identifies the control.

The *x* and *y* fields take integer values that specify the *x* and *y* coordinates of the upper-left corner of the control. The horizontal units are 1/4 of the dialog base width unit; the vertical units are 1/8 of the dialog base height unit. The current dialog base units are computed from the height and width of the current system font. The **GetDialogBaseUnits** function returns the dialog base units in pixels. The coordinates are relative to the origin of the dialog box.

The *width* and *height* fields take integer values that specify the width and height of the control. The width units are 1/4 of the dialog base width unit; the height units are 1/8 of the dialog base height unit.

The optional *style* field can contain any combination (or none) of the following styles:

- WS_TABSTOP
- WS_GROUP
- WS_VSCROLL
- WS_HSCROLL
- WS_DISABLED

These styles are described in Table 8.1, "Window Styles."

In addition to these styles, the *style* field may contain any combination (or none) of the EDIT-class styles described in Table 8.3, "Control Styles." Styles can be combined using the bitwise OR operator. The EDIT-class styles must not conflict with each other.

Comments

The *x*, *y*, *width*, and *height* fields can use the addition operator (+) for relative positioning. For example, "15 + 6" can be used for the *x* field.

The default style for **EDITTEXT** is WS_TABSTOP, ES_LEFT, and WS_BORDER.

Keyboard use is predefined for edit controls. Predefined keys are listed in the *System Application Architecture, Common User Access: Advanced Interface Design Guide.*

The following example demonstrates the correct usage of the **EDITTEXT** statement:

```
EDITTEXT  3, 10, 10, 100, 10
```

COMBOBOX Statement

Syntax

COMBOBOX *id*, *x*, *y*, *width*, *height*, [[*style*]]

This statement defines a combo box belonging to the COMBOBOX class. A combo box consists of either a static text field or edit field combined with a list box. The list box can be displayed at all times or pulled down by the user. If the combo box contains a static text field, the text field always displays the selection (if any) in the list-box portion of the combo box. If it uses an edit field, the user can type in the desired selection; the list box highlights the first item (if any) which matches what the user has entered in the edit field. The user can then select the item highlighted in the list box to complete the choice. In addition, the combo box can be owner-draw and of fixed or variable height.

The *id* field takes a unique integer value that identifies the control.

The *x* and *y* fields take integer values that specify the *x* and *y* coordinates of the upper-left corner of the control. The horizontal units are ¼ of the dialog base width unit; the vertical units are ⅛ of the dialog base height unit. The current dialog base units are computed from the height and width of the current system font. The **GetDialogBaseUnits** function returns the dialog base units in pixels. The coordinates are relative to the origin of the dialog box.

The *width* and *height* fields take integer values that specify the width and height of the control. The width units are ¼ of the dialog base width unit; the height units are ⅛ of the dialog base height unit.

The optional *style* field can contain any combination (or none) of the following styles:

- WS_TABSTOP
- WS_GROUP
- WS_VSCROLL
- WS_DISABLED

These styles are described in Table 8.1, "Window Styles."

In addition to these styles, the *style* field may contain any combination (or none) of the combo-box styles described in Table 8.3, "Control Styles." Styles can be combined using the bitwise OR operator.

Comments

The *x*, *y*, *width*, and *height* fields can use the addition operator (+) for relative positioning. For example, "15 + 6" can be used for the *x* field.

The default style for **COMBOBOX** is WS_TABSTOP and CBS_SIMPLE.

For information on the recommended keys for use in combo-box controls, see the *System Application Architecture, Common User Access: Advanced Interface Design Guide*.

The following example demonstrates the correct usage of the **COMBOBOX** statement:

```
COMBOBOX 777, 10, 10, 50, 54, CBS_SIMPLE | WS_VSCROLL | WS_TABSTOP
```

ICON Statement

Syntax

ICON *text, id, x, y, width, height,* [[*style*]]

This statement defines an icon control belonging to the STATIC class. It creates an icon displayed in the dialog box.

The *text* field specifies the name of an icon (not a filename) defined elsewhere in the resource file.

The *id* field takes a unique integer value that identifies the control.

The *x* and *y* fields take integer values that specify the *x* and *y* coordinates of the upper-left corner of the control. The horizontal units are ¼ of the dialog base width unit; the vertical units are ⅛ of the dialog base height unit. The current dialog base units are computed from the height and width of the current system font. The **GetDialogBaseUnits** function returns the dialog base units in pixels. The coordinates are relative to the origin of the dialog box.

For the **ICON** statement, the *width* and *height* fields are ignored; the icon automatically sizes itself.

The optional *style* field allows only the SS_ICON style.

Comments

The *x, y, width,* and *height* fields can use the addition operator (+) for relative positioning. For example, "15 + 6" can be used for the *x* field.

The default style for **ICON** is SS_ICON.

The following example demonstrates the correct usage of the **ICON** statement:

```
ICON "myicon" 901, 30, 30
```

SCROLLBAR Statement

Syntax

SCROLLBAR *id, x, y, width, height,* [[*style*]]

This statement defines a scroll-bar control belonging to the SCROLLBAR class. It is a rectangle that contains a scroll thumb and has direction arrows at both ends. The scroll-bar control sends a notification message to its parent whenever the user clicks the mouse in the control. The parent is responsible for updating the thumb position. Scroll-bar controls can be positioned anywhere in a window and used whenever needed to provide scrolling input..

The *id* field takes a unique integer value that identifies the control.

The *x* and *y* fields take integer values that specify the location of the upper-left corner of the control in dialog units relative to the origin of the dialog box. The horizontal units are ¼ of the dialog base width unit; the vertical units are ⅛ of the dialog base height unit. The current dialog base units are computed from the height and width of the current system font. The **GetDialogBaseUnits** function returns the dialog base units in pixels.

The *width* and *height* fields take integer values that specify the width and height of the control. The width units are ¼ of the dialog base width unit; the height units are ⅛ of the dialog base height unit.

The optional *style* field can contain any combination (or none) of the following styles:

- WS_TABSTOP
- WS_GROUP
- WS_DISABLED

These styles are described in Table 8.1, "Window Styles."

In addition to these styles, the *style* field may contain any combination (or none) of the SCROLLBAR-class styles described in Table 8.3, "Control Styles." Styles can be combined using the bitwise OR operator.

Comments

The *x, y, width,* and *height* fields can use the addition operator (+) for relative positioning. For example, "15 + 6" can be used for the *x* field.

The default style for SCROLLBAR is SBS_HORZ.

The following example demonstrates the correct usage of the **SCROLLBAR** statement:

```
SCROLLBAR 999, 25, 30, 10, 100
```

CONTROL Statement

Syntax

CONTROL *text, id, class, style, x, y, width, height*

This statement defines a user-defined control window.

The *text* field takes an ASCII string that specifies the text to be displayed. The string must be enclosed in double quotation marks.

The *id* field takes a unique integer value that identifies the control.

The *class* field takes a predefined name, character string, or integer value that defines the class. This can be any one of the control classes; for a list of the control classes, see Table 8.2, "Control Classes." If the value is a predefined name supplied by the application, it must be an ASCII string enclosed in double quotation marks.

The *style* field takes a predefined name or integer value that specifies the style of the given control. The exact meaning of *style* depends on the *class* value. Tables 8.2, "Control Classes," and 8.3, "Control Styles," list the control classes and corresponding styles.

The *x* and *y* fields take integer values that specify the *x* and *y* coordinates of the upper-left corner of the control. The horizontal units are 1/4 of the dialog base width unit; the vertical units are 1/8 of the dialog base height unit. The current dialog base units are computed from the height and width of the current system font. The **GetDialogBaseUnits** function returns the dialog base units in pixels. The coordinates are relative to the origin of the dialog box.

The *width* and *height* fields take integer values that specify the width and height of the control. The width units are 1/4 of the dialog base width unit; the height units are 1/8 of the dialog base height unit.

Comments

The *x*, *y*, *width*, and *height* fields can use the addition operator (+) for relative positioning. For example, "15 + 6" can be used for the *x* field.

Table 8.2 describes the six control classes:

Table 8.2 Control Classes

Class	Description
BUTTON	A button control is a small rectangular child window that represents a "button" that the user can turn on or off by clicking it with the mouse. Button controls can be used alone or in groups, and can either be labeled or appear without text. Button controls typically change appearance when the user clicks them.
COMBOBOX	Combo-box controls consist of a selection field similar to an edit control plus a list box. The list box may be displayed at all times or may be dropped down when the user selects a "pop box" next to the selection field. Depending on the style of the combo box, the user can or cannot edit the contents of the selection field. If the list box is visible, typing characters into the selection box will cause the first list box entry which matches the characters typed to be highlighted. Conversely, selecting an item in the list box displays the selected text in the selection field.

Table 8.2 **Control Classes** *(continued)*

Class	Description
EDIT	An edit control is a rectangular child window in which the user can enter text from the keyboard. The user selects the control, and gives it the input focus, by clicking the mouse inside it or pressing the TAB key. The user can enter text when the control displays a flashing caret. The mouse can be used to move the cursor and select characters to be replaced, or to position the cursor for inserting characters. The BACKSPACE key can be used to delete characters. Edit controls use the fixed-pitch font and display ANSI characters. They expand tab characters into as many space characters as are required to move the cursor to the next tab stop. Tab stops are assumed to be at every eighth character position.
LISTBOX	List-box controls consist of a list of character strings. The control is used whenever an application needs to present a list of names, such as filenames, that the user can view and select. The user can select a string by pointing to the string with the mouse and clicking a mouse button. When a string is selected, it is highlighted, and a notification message is passed to the parent window. A scroll bar can be used with a list-box control to scroll lists that are too long or too wide for the control window.
SCROLLBAR	A scroll-bar control is a rectangle that contains a scroll thumb and has direction arrows at both ends. The scroll bar sends a notification message to its parent whenever the user clicks the mouse in the control. The parent is responsible for updating the thumb position, if necessary. Scroll-bar controls have the same appearance and function as the scroll bars used in ordinary windows. But unlike scroll bars, scroll-bar controls can be positioned anywhere within a window and used whenever needed to provide scrolling input for a window. The scroll-bar class also includes size-box controls. A size-box control is a small rectangle that the user can expand to change the size of the window.
STATIC	Static controls are simple text fields, boxes, and rectangles that can be used to label, box, or separate other controls. Static controls take no input and provide no output.

Table 8.3 describes the control styles for each of the control classes:

Table 8.3 Control Styles

Style	Description
BUTTON Class	
BS_PUSHBUTTON	A small elliptical button containing the given text. The control sends a message to its parent whenever the user clicks the mouse inside the rectangle.
BS_DEFPUSHBUTTON	A small elliptical button with a bold border. This button represents the default user response. Any text is displayed within the button. Windows sends a message to the parent window when the user clicks the mouse in this button.
BS_CHECKBOX	A small rectangular button that can be checked; its border becomes bold when the user clicks the mouse in it. Any text appears to the right of the button.
BS_AUTOCHECKBOX	Identical to BS_CHECKBOX except that the button automatically toggles its state whenever the user clicks it.
BS_RADIOBUTTON	A small circular button whose border becomes bold when the user clicks the mouse in it. In addition, to make the border bold, Windows sends a message to the button's parent notifying it that a click occurred. On the next click, Windows makes the border normal again and sends another message.
BS_AUTORADIOBUTTON	Identical to BS_RADIOBUTTON except that when the button is checked, the application is notified with BN_CLICKED, and all other radio buttons in the group are unchecked.
BS_LEFTTEXT	Text appears on the left side of the radio button or check-box button. Use this style with BS_CHECKBOX, BS_3STATE, or BS_RADIOBUTTON styles.

Table 8.3 Control Styles *(continued)*

Style	Description
BS_3STATE	Identical to BS_CHECKBOX except that a button can be grayed as well as checked or unchecked. The grayed state is typically used to show that a check box has been disabled.
BS_AUTO3STATE	Identical to BS_3STATE except that the button automatically toggles its state when the user clicks it.
BS_GROUPBOX	A rectangle into which other buttons are grouped. Any text is displayed in the rectangle's upper-left corner.
BS_OWNERDRAW	An owner-draw button. The parent window is notified when the button is clicked. Notification includes a request to paint, invert, and disable the button.
COMBOBOX Class	
CBS_SIMPLE	Displays the list box at all times. The current selection in the list box is displayed in the edit control.
CBS_DROPDOWN	Is similar to CBS_SIMPLE, except that the list box is not displayed unless the user selects an icon next to the selection field.
CBS_DROPDOWNLIST	Is similar to CBS_DROPDOWN, except that the edit control is replaced by a static text item which displays the current selection in the list box.
CBS_OWNERDRAWFIXED	Specifies a fixed-height owner-draw combo box. The owner of the list box is responsible for drawing its contents; the items in the list box are all the same height.
CBS_OWNERDRAWVARIABLE	Specifies a variable-height owner-draw combo box. The owner of the list box is responsible for drawing its contents; the items in the list box can have different heights.

Table 8.3 Control Styles *(continued)*

Style	Description
CBS_AUTOHSCROLL	Scrolls the text in the edit control to the right when the user types a character at the end of the line. If this style is not set, only text which fits within the rectangular boundary is allowed.
CBS_SORT	Sorts strings entered into the list box.
CBS_HASSTRINGS	Specifies an owner-draw combo box that contains items consisting of strings. The combo box maintains the memory and pointers for the strings so that the application can use the LB_GETTEXT message to retrieve the text for a particular item.
CBS_OEMCONVERT	Text entered in the combo box edit control is converted from the ANSI character set to the OEM character set and then back to ANSI. This ensures proper character conversion when the application calls the **AnsiToOem** function to convert an ANSI string in the combo box to OEM characters. This style is most useful for combo boxes that contain filenames and applies only to combo boxes created with the CBS_SIMPLE or CBS_DROPDOWN styles.
EDIT Class	
ES_LEFT	Flush-left text.
ES_CENTER	Centered text. This style is valid in multiline edit controls only.
ES_RIGHT	Flush-right text. This style is valid in multiline edit controls only.
ES_LOWERCASE	Lowercase edit control. An edit control with this style converts all characters to lowercase as they are typed into the edit control.

Table 8.3 Control Styles *(continued)*

Style	Description
ES_UPPERCASE	Uppercase edit control. An edit control with this style converts all characters to uppercase as they are typed into the edit control.
ES_PASSWORD	Password edit control. An edit control with this style displays all characters as an asterisk (*) as they are typed into the edit control. An application can use the EM_SETPASSWORDCHAR message to change the character that is displayed.
ES_MULTILINE	Multiple-line edit control. (The default is single-line.) If the ES_AUTOVSCROLL style is specified, the edit control shows as many lines as possible and scrolls vertically when the user presses the ENTER key. (This is actually the carriage-return character, which the edit control expands to a carriage-return/line-feed combination. A line feed is not treated the same as a carriage return.) If ES_AUTOVSCROLL is not given, the edit control shows as many lines as possible and beeps if the user presses ENTER when no more lines can be displayed. If the ES_AUTOHSCROLL style is specified, the multiple-line edit control automatically scrolls horizontally when the caret goes past the right edge of the control. To start a new line, the user must press the ENTER key. If ES_AUTOHSCROLL is not given, the control automatically wraps words to the beginning of the next line when necessary; a new line is also started if the user presses ENTER. The position of the wordwrap is determined by the window size. If the window size changes, the wordwrap position changes, and the text is redisplayed.

Table 8.3 Control Styles *(continued)*

Style	Description
	Multiple-line edit controls can have scroll bars. An edit control with scroll bars processes its own scroll-bar messages. Edit controls without scroll bars scroll as described above, and process any scroll messages sent by the parent window.
ES_AUTOVSCROLL	Text is automatically scrolled up one page when the user presses the ENTER key on the last line.
ES_AUTOHSCROLL	Text is automatically scrolled to the right by 10 characters when the user types a character at the end of the line. When the user presses the ENTER key, the control scrolls all text back to position 0.
ES_NOHIDESEL	Normally, an edit control hides the selection when the control loses the input focus, and inverts the selection when the control receives the input focus. Specifying ES_NOHIDESEL overrides this default action.
ES_OEMCONVERT	Text entered in the edit control is converted from the ANSI character set to the OEM character set and then back to ANSI. This ensures proper character conversion when the application calls the **AnsiToOem** function to convert an ANSI string in the edit control to OEM characters. This style is most useful for edit controls that contain filenames.
LISTBOX Class	
LBS_STANDARD	Strings in the list box are sorted alphabetically and the parent window receives an input message whenever the user clicks or double-clicks a string. The list box contains borders on all sides.

Table 8.3 Control Styles *(continued)*

Style	Description
LBS_EXTENDEDSEL	The user can select multiple items using the mouse with the SHIFT and/or the CONTROL key or special key combinations.
LBS_HASSTRINGS	An owner-draw list box contains items consisting of strings. The list box maintains the memory and pointers for the strings so the application can use the LB_GETTEXT message to retrieve the text for a particular item.
LBS_NOTIFY	The parent receives an input message whenever the user clicks or double-clicks a string.
LBS_MULTIPLESEL	The string selection is toggled each time the user clicks or double-clicks the string. Any number of strings can be selected.
LBS_MULTICOLUMN	The list box contains multiple columns. The list box can be scrolled horizontally. The LB_SETCOLUMNWIDTH message sets the width of the columns.
LBS_NOINTEGRALHEIGHT	The size of the list box is exactly the size specified by the application when it created the list box. Normally, Windows sizes a list box so that the list box does not display partial items.
LBS_SORT	The strings in the list box are sorted alphabetically.
LBS_NOREDRAW	The list-box display is not updated when changes are made. This style can be changed at any time by sending a WM_SETREDRAW message.
LBS_OWNERDRAWFIXED	The owner of the list box is responsible for drawing its contents; the items in the list box are all the same height.
LBS_OWNERDRAWVARIABLE	The owner of the list box is responsible for drawing its contents; the items in the list box are variable in height.

Table 8.3 Control Styles (*continued*)

Style	Description
LBS_USETABSTOPS	The list box is able to recognize and expand tab characters when drawing its strings. The default tab positions are set at every 32 dialog units. (A dialog unit is a horizontal or vertical distance. One horizontal dialog unit is equal to ¼ of the current dialog base width unit. The dialog base units are computed from the height and width of the current system font. The **GetDialogBaseUnits** function returns the size of the dialog base units in pixels.)
LBS_WANTKEYBOARDINPUT	The owner of the list box receives WM_VKEYTOITEM or WM_CHARTOITEM messages whenever the user presses a key when the list box has input focus. This allows an application to perform special processing on the keyboard input.
SCROLLBAR Class	
SBS_VERT	Vertical scroll bar. If neither SBS_RIGHTALIGN nor SBS_LEFTALIGN is specified, the scroll bar has the height, width, and position given in the **CreateWindow** function.
SBS_RIGHTALIGN	Used with SBS_VERT. The right edge of the scroll bar is aligned with the right edge of the rectangle specified by the *x*, *y*, *width*, and *height* values given in the **CreateWindow** function. The scroll bar has the default width for system scroll bars.
SBS_LEFTALIGN	Used with SBS_VERT. The left edge of the scroll bar is aligned with the left edge of the rectangle specified by the *x*, *y*, *width*, and *height* values given in the **CreateWindow** function. The scroll bar has the default width for system scroll bars.

Table 8.3 Control Styles *(continued)*

Style	Description
SBS_HORZ	Horizontal scroll bar. If neither SBS_BOTTOMALIGN nor SBS_TOPALIGN is specified, the scroll bar has the height, width, and position given in the **CreateWindow** function.
SBS_TOPALIGN	Used with SBS_HORZ. The top edge of the scroll bar is aligned with the top edge of the rectangle specified by the *x*, *y*, *width*, and *height* values given in the **CreateWindow** function. The scroll bar has the default height for system scroll bars.
SBS_BOTTOMALIGN	Used with SBS_HORZ. The bottom edge of the scroll bar is aligned with the bottom edge of the rectangle specified by the *x*, *y*, *width*, and *height* values given in the **CreateWindow** function. The scroll bar has the default height for system scroll bars.
SBS_SIZEBOX	Size box. If neither SBS_SIZEBOXBOTTOMRIGHTALIGN nor SBS_SIZEBOXTOPLEFTALIGN is specified, the size box has the height, width, and position given in the **CreateWindow** function.
SBS_SIZEBOXTOPLEFTALIGN	Used with SBS_SIZEBOX. The top-left corner of the size box is aligned with the top-left corner of the rectangle specified by the *x*, *y*, *width*, and *height* values given in the **CreateWindow** function. The size box has the default size for system size boxes.
SBS_SIZEBOXBOTTOMRIGHTALIGN	Used with SBS_SIZEBOX. The bottom-right corner of the size box is aligned with the bottom-right corner of the rectangle specified by the *x*, *y*, *width*, and *height* values given in the **CreateWindow** function. The size box has the default size for system size boxes.

Table 8.3 Control Styles *(continued)*

Style	Description
STATIC Class	
SS_LEFT	A simple rectangle displaying the given text flush left in the rectangle. The text is formatted before it is displayed. Words that would extend past the end of a line are automatically wrapped to the beginning of the next line.
SS_CENTER	A simple rectangle displaying the given text centered in the rectangle. The text is formatted before it is displayed. Words that would extend past the end of a line are automatically wrapped to the beginning of the next line.
SS_RIGHT	A simple rectangle displaying the given text flush right in the rectangle. The text is formatted before it is displayed. Words that would extend past the end of a line are automatically wrapped to the beginning of the next line.
SS_LEFTNOWORDWRAP	A simple rectangle displaying the given text flush left in the rectangle. Tabs are expanded, but words are not wrapped. Text that extends past the end of a line is clipped.
SS_SIMPLE	Designates a simple rectangle and displays a single line of text flush-left in the rectangle. The line of text cannot be shortened or altered in any way. (The control's parent window or dialog box must not process the WM_CTLCOLOR message.)

Table 8.3 Control Styles *(continued)*

Style	Description
SS_NOPREFIX	Unless this style is specified, windows will interpret any "&" characters in the control's text to be accelerator prefix characters. In this case, the "&" is removed and the next character in the string is underlined. If a static control is to contain text where this feature is not wanted, SS_NOPREFIX may be added. This static-control style may be included with any of the defined static controls. You can combine SS_NOPREFIX with other styles by using the bitwise OR operator. This is most often used when filenames or other strings that may contain an "&" need to be displayed in a static control in a dialog box.
SS_ICON	An icon displayed in the dialog box. The given text is the name of an icon (not a filename) defined elsewhere in the resource file. For the **ICON** statement, the *width* and *height* parameters in the **CreateWindow** function are ignored; the icon automatically sizes itself.
SS_BLACKRECT	A rectangle filled with the color used to draw window frames. This color is black in the default Windows color scheme.
SS_GRAYRECT	A rectangle filled with the color used to fill the screen background. This color is gray in the default Windows color scheme.
SS_WHITERECT	A rectangle filled with the color used to fill window backgrounds. This color is white in the default Windows color scheme.
SS_BLACKFRAME	Box with a frame drawn with the same color as window frames. This color is black in the default Windows color scheme.

Table 8.3 Control Styles (*continued*)

Style	Description
SS_GRAYFRAME	Box with a frame drawn with the same color as the screen background (desktop). This color is gray in the default Windows color scheme.
SS_WHITEFRAME	Box with a frame drawn with the same color as window backgrounds. This color is white in the default Windows color scheme.
SS_USERITEM	User-defined item.

8.8 Directives

The resource directives are special statements that define actions to be performed on the script file before it is compiled. The directives can assign values to names, include the contents of files, and control compilation of the script file.

The resource directives are identical to the directives used in the C programming language. They are fully defined in the *Microsoft C Language Reference.*

8.8.1 #include Statement

Syntax

#include *filename*

This directive copies the contents of the file specified by *filename* into your resource script before the Resource Compiler processes the script. It replaces the **rcinclude** directive of versions prior to Windows 3.0.

The *filename* field is an ASCII string that specifies the DOS filename of the file to be included, using the same syntax as the C-language preprocessor **#include** directive. A forward slash (/) can be used instead of a backslash (for example, "root/sub"). If the filename has the .H or .C extension, only the preprocessor directives in the file are processed. Otherwise, this directive processes the entire contents of the file.

The following example demonstrates the correct usage of the **#include** statement:

```
#include "WINDOWS.H"

PenSelect MENU
BEGIN
    Menuitem "&Black pen", BLACK_PEN
END
```

8.8.2 #define Statement

Syntax

#define *name value*

This directive assigns the given *value* to *name*. All subsequent occurrences of *name* are replaced by *value*.

The *value* field takes any integer value, character string, or line of text.

The following example demonstrates the correct usage of the **#define** statement:

```
#define     nonzero       1
#define     USERCLASS   "MyControlClass"
```

8.8.3 #undef Statement

Syntax

#undef *name*

This directive removes the current definition of *name*. All subsequent occurrences of *name* are processed without replacement.

The following example demonstrates the correct usage of the **#undef** statement:

```
#undef     nonzero
#undef     USERCLASS
```

8.8.4 #ifdef Statement

Syntax

#ifdef *name*

This directive carries out conditional compilation of the resource file by checking the specified *name*. If *name* has been defined using a **#define** directive, **#ifdef** directs the Resource Compiler to continue with the statement immediately after **#ifdef**. If *name* has not been defined, **#ifdef** directs the compiler to skip all statements up to the next **#endif** directive.

Thr following example demonstrates the correct usage of the **#ifdef** statement:

```
#ifdef Debug
errbox BITMAP errbox.bmp
#endif
```

8.8.5 #ifndef Statement

Syntax

#ifndef *name*

This directive carries out conditional compilation of the resource file by checking the specified *name*. If *name* has not been defined or if its definition has been removed using the **#undef** directive, **#ifndef** directs the Resource Compiler to continue processing statements up to the next **#endif, #else**, or **#elif** directive, and then to skip to the statement after **#endif.** If *name* is defined, **#ifndef** directs the compiler to skip to the next **#endif, #else**, or **#elif** directive.

The following example demonstrates the correct usage of the **#ifndef** statement:

```
#ifndef Optimize
errbox BITMAP errbox.bmp
#endif
```

8.8.6 #if Statement

Syntax

#if *constant-expression*

This directive carries out conditional compilation of the resource file by checking the specified *constant-expression.* If *constant-expression* is nonzero, **#if** directs the Resource Compiler to continue processing statements up to the next **#endif, #else**, or **#elif** directive, then skip to the statement after **#endif.** If *constant-expression* is zero, **#if** directs the compiler to skip to the next **#endif, #else**, or **#elif** directive.

The *constant-expression* field specifies a defined name, an integer constant, or an expression consisting of names, integers, and arithmetical and relational operators.

The following example demonstrates the correct usage of the **#if** statement:

```
#if Version<3
errbox BITMAP errbox.bmp
#endif
```

8.8.7 #elif Statement

Syntax

#elif *constant-expression*

This directive marks an optional clause of a conditional compilation block defined by an **#ifdef, #ifndef,** or **#if** directive. The **#elif** directive carries out conditional compilation of the resource file by checking the specified *constant-expression.* If *constant-expression* is nonzero, **#elif** directs the Resource Compiler to continue processing statements up to the next **#endif, #else,** or **#elif** directive, then skip to the statement after **#endif.** If *constant-expression* is zero, **#elif** directs the compiler to skip to the next **#endif, #else,** or **#elif** directive. Any number of **#elif** directives can be used in a conditional block.

The *constant-expression* field specifies a defined name, an integer constant, or an expression consisting of names, integers, and arithmetical and relational operators.

The following demonstrates the correct usage of the **#elif** statement:

```
#if Version<3
errbox BITMAP errbox.bmp
#elif Version<7
errbox BITMAP userbox.bmp
#endif
```

8.8.8 #else Statement

Syntax

#else

This directive marks an optional clause of a conditional compilation block defined by an **#ifdef, #ifndef,** or **#if** directive. The **#else** directive must be the last directive before **#endif.**

The following example demonstrates the correct usage of the **#else** statement:

```
#ifdef Debug
errbox BITMAP errbox.bmp
#else
errbox BITMAP userbox.bmp
#endif
```

8.8.9 #endif Statement

Syntax

#endif

This directive marks the end of a conditional compilation block defined by an **#if** or **#ifdef** directive. One **#if** or **#endif** is required for each **#ifdef** directive.

8.9 Summary

The resource script statements define resources that the Resource Compiler adds to an application's executable file. The application can then load these resources as they are needed at run time. For more information on topics related to the Resource Compiler, see the following:

Topic	Reference
General information on Windows programming	*Guide to Programming*: Chapter 1, "An Overview of the Windows Environment"
Menus	*Guide to Programming*: Chapter 7, "Menus"
Controls and dialog boxes	*Guide to Programming*: Chapter 8, "Controls," and Chapter 9, "Dialog Boxes"
Using the Resource Compiler	*Tools*: Chapter 3, "Compiling Resources: The Resource Compiler"
Designing dialog boxes	*Tools*: Chapter 5, "Designing Dialog Boxes: The Dialog Editor"

Chapter 9 File Formats

This chapter describes the file formats used to create, execute, and supply data to Microsoft Windows applications. These files include the following:

- Bitmap files
- Icon resource files
- Cursor resource files
- Clipboard files
- Metafiles

9.1 Bitmap File Formats

Windows version 3.0 bitmap files store a bitmap in a device-independent format which allows Windows to display the bitmap on any device. In this case, the term "device independent" means that the bitmap specifies pixel color in a form independent of the method used by any particular device to represent color. The assumed file extension of a Windows device-independent bitmap file is .BMP.

Each bitmap file contains a **BITMAPFILEHEADER** data structure immediately followed by a single, device-independent bitmap (DIB) consisting of a **BITMAPINFO** data structure and an array of bytes that defines the bitmap bits.

Windows version 3.0 also reads bitmap files in the format read by Microsoft OS/2 Presentation Manager version 1.2. These files consist of a **BITMAPFILEHEADER** data structure immediately followed by a **BITMAPCOREINFO** data structure. Following this data structure is an array of bytes that defines the bitmap bits.

See Chapter 7, "Data Types and Structures," for information on the **BITMAPFILEHEADER**, **BITMAPCOREINFO** and **BITMAPINFO** data structures.

9.2 Icon Resource File Format

An icon resource file (with the .ICO file extension) can be device independent both for color and resolution.

Icon resource files can contain multiple device-independent bitmaps defining the icon image, one for each targeted display-device resolution. Windows detects the resolution of the current display and matches it against the *x* and *y* pixel-size values specified for each version of the image. If Windows determines that there is an exact match between an icon image and the current device, then it uses the matching image; otherwise, it selects the closest match and stretches the image to the proper size.

If an icon resource file contains more than one image for a particular resolution, Windows uses the icon image that most closely matches the color capabilities of the current display device. If no image exists which exactly matches the device capabilities, Windows selects the image which has the greatest number of colors without exceeding the number of display-device colors. If all images exceed the color capabilities of the current display device, then Windows uses the icon image with the least number of colors.

The icon resource file contains a header structure at the beginning of the file which identifies the type and number of icon images contained in the file. The following shows the format of this header:

Field	Type/Description
icoReserved	**WORD** Is reserved and must be set to 0.
icoResourceType	**WORD** Specifies the type of resource contained in the file. For an icon resource, this field must be 1.
icoResourceCount	**WORD** Specifies the number of images contained in the file.

The resource directory follows this header. The resource directory consists of one or more arrays of resource descriptors. The **icoResorceCount** specifies the number of arrays. The following list shows the format of the array:

Field	Type/Description
Width	**BYTE** Specifies the width in pixels of this form of the icon image. Acceptable values are 16, 32, or 64.
Height	**BYTE** Specifies the height in pixels of this form of the icon image. Acceptable values are 16, 32, or 64.

Field	Type/Description
ColorCount	**BYTE** Specifies the number of colors in this form of the icon image. Acceptable values are 2, 8, or 16.
Reserved	**BYTE** Reserved for future use.
Reserved	**WORD** Reserved for future use.
Reserved	**WORD** Reserved for future use.
icoDIBSize	**DWORD** Specifies in bytes the size of the pixel array for this form of the icon image.
icoDIBOffset	**DWORD** Specifies the offset in bytes from the beginning of the file to the device-independent bitmap for this form.

Icons can be in color. To achieve transparency, the DIB for each icon will consist of two parts:

1. The first part is a color bitmap which supplies the XOR mask for the icon.
2. The second part is a monochrome bitmap which provides the AND mask that defines the transparent portion of the icon.

The monochrome bitmap does not contain a DIB header, but instead immediately follows the color bitmap. It must have the same pixel height as the color bitmap.

9.3 Cursor Resource File Format

Like icon resource files, cursor resource files (with the .CUR file extension) may contain multiple images to match targeted display-device resolutions. In the case of cursors, Windows determines the best match for a particular display-device driver by examining the width and height of the cursor images.

The cursor resource file contains a header structure at the beginning of the file which identifies the type and number of resources in the file. The following shows the format of this header:

Field	Type/Description
curReserved	**WORD** Is reserved and must be set to 0.
curResourceType	**WORD** Specifies the type of resource contained in the file. For a cursor resource, this field must be 2.
curResourceCount	**WORD** Specifies the number of resources contained in the file.

The resource directory follows this header. The resource directory consists of one or more arrays of resource descriptors. The **curResorceCount** specifies the number of arrays. The following shows the format of the array:

Field	Type/Description
curWidth	**BYTE** Specifies the width in pixels of this form of the cursor image.
curHeight	**BYTE** Specifies the height in pixels of this form of the cursor image.
ColorCount	**BYTE** Specifies the number of colors in this form of the icon image. Acceptable values are 2, 8, or 16.
Reserved	**BYTE** Is reserved and must be set to 0.
curXHotspot	**WORD** Specifies in pixels the horizontal position of the hotspot.
curYHotspot	**WORD** Specifies in pixels the vertical position of the hotspot.
curDIBSize	**DWORD** Specifies in bytes the size of the pixel array for this form of the cursor image.
curDIBOffset	**DWORD** Specifies in bytes the offset to the device-independent bitmap for this form. The offset is from the beginning of the file.

Cursors are monochrome. The bitmap for a cursor consists of two parts; the first half is the XOR mask specifying the visible image, and the second half is the AND mask specifying the transparent portion of the cursor image. The two parts must be of equal width and height. By combining the values in corresponding mask bits, Windows determines whether a pixel is black, white, inverted, or transparent.

Table 9.1 shows what values are necessary to produce the corresponding colors, inversions, or transparencies:

Table 9.1 Bit Mask Results

	Bit Value	Bit Value	Bit Value	Bit Value
AND mask	0	0	1	1
XOR mask	0	1	0	1
Resultant Pixel	Black	White	Transparent	Inverted

Figure 9.1 shows two bitmaps that represent the AND mask and the XOR mask for a cursor. The bit settings in the two bitmaps create a black, cross-shaped cursor:

AND Mask

1	0	1
0	0	0
1	0	1

XOR Mask

0	0	0
0	0	0
0	0	0

Resultant Cursor

Figure 9.1 Settings and Resultant Cursor

9.4 Clipboard File Format

The Windows clipboard saves and reads clipboard data in files with the .CLP extension. A clipboard-data file contains a value that identifies it as a clipboard-data file, one or more data structures defining the format, size, and location of the clipboard data, and one or more blocks of the actual data.

The clipboard-data file begins with a header consisting of two fields. The following describes these fields:

Field	Type/Description
FileIdentifier	**WORD** Identifies the file as a clipboard-data file. This field must be set to CLP_ID.
FormatCount	**WORD** Specifies the number of clipboard formats contained in the file.

This header is followed by one or more data structures, each of which identifies the format, size, and offset of a block of clipboard data. The following shows the fields of this data structure:

Field	Type/Description
FormatID	**WORD** Specifies the clipboard-format ID of the clipboard data. See the description of the **SetClipboardData** function in Chapter 4, "Functions Directory," in *Reference, Volume 1*, for information on clipboard formats.
LenData	**DWORD** Specifies in bytes the length of the clipboard data.

Field	Type/Description
OffData	**DWORD** Specifies in bytes the offset of the clipboard-data block. The offset is from the beginning of the file.
Name	Is a 79-character array that specifies the format name for a private clipboard format.

The first block of clipboard data follows the last of these structures. For bitmaps and metafiles, the bits follow immediately after the bitmap header and the **METAFILEPICT** data structures.

9.5 Metafile Format

A metafile consists of a collection of graphics device interface (GDI) function calls that create specific images on a device. Metafiles provide convenient storage for images that appear repeatedly in applications, and also allow you to use the clipboard to cut and paste images from one application to another.

Metafiles store images as a series of GDI function calls. After storing the function calls, applications play a metafile to generate an image on a device.

When an object is created during playback, GDI adds the handle of the object to the first available entry in the metafile handle table. GDI clears the table entry corresponding to the object when it is deleted during playback, allowing the table entry to be reused when another object is created.

NOTE Functions described in this section are discussed in greater detail in Chapter 4, "Functions Directory," in *Reference, Volume 1.*

The metafile itself consists of two parts: a header and a list of records. The header contains a description of the size (in words) of the metafile and the number of drawing objects it uses. The list of records contains the GDI functions. The drawing objects can be pens, brushes, or bitmaps.

9.5.1 Metafile Header

The following structured list shows the format of the metafile header:

```
struct{
 WORD  mtType;
 WORD  mtHeaderSize;
 WORD  mtVersion;
 DWORD mtSize;
 WORD  mtNoObjects;
 DWORD mtMaxRecord;
```

```
 WORD  mtNoParameters;
}
```

The metafile header contains the following fields:

Field	Description
mtType	Specifies whether the metafile is in memory or recorded in a disk file. It is one of these two values: **Value** — **Meaning** 1 — Metafile is in memory 2 — Metafile is in a disk file
mtHeaderSize	Specifies the size in words of the metafile header.
mtVersion	Specifies the Windows version number. The version number for Windows version 3.0 is 0x300.
mtSize	Specifies the size in words of the file.
mtNoObjects	Specifies the maximum number of objects that exist in the metafile at the same time.
mtMaxRecord	Specifies the size in words of the largest record in the metafile.
mtNoParameters	Is not used.

9.5.2 Metafile Records

A series of records follows the metafile header. Metafile records describe GDI functions. GDI stores most of the GDI functions that an application can use to create metafiles in similar, typical records. "Typical Metafile Record," later in this section, shows the format of the typical metafile record. Table 9.2, "GDI Functions and Values," lists the functions which GDI records in typical records, along with their respective function numbers.

The remainder of the functions contain more complex structures in their records. "Function-Specific Records," later in this section, describes the records for these functions.

In some cases, there are two versions of a metafile record. One version represents the record created by versions of Windows prior to version 3.0, while the second version represents the record created by Windows versions 3.0 and later. Windows 3.0 plays all metafile versions, but stores only 3.0 versions. Windows versions prior to 3.0 will not play metafiles recorded by Windows 3.0.

Table 9.2 GDI Functions and Values

Function	Value
Arc	0x0817
Chord	0x0830
Ellipse	0x0418
ExcludeClipRect	0x0415
FloodFill	0x0419
IntersectClipRect	0x0416
LineTo	0x0213
MoveTo	0x0214
OffsetClipRgn	0x0220
OffsetViewportOrg	0x0211
OffsetWindowOrg	0x020F
PatBlt	0x061D
Pie	0x081A
RealizePalette (3.0 and later)	0x0035
Rectangle	0x041B
ResizePalette (3.0 and later)	0x0139
RestoreDC	0x0127
RoundRect	0x061C
SaveDC	0x001E
ScaleViewportExt	0x0412
ScaleWindowExt	0x0400
SetBkColor	0x0201
SetBkMode	0x0102
SetMapMode	0x0103
SetMapperFlags	0x0231
SetPixel	0x041F
SetPolyFillMode	0x0106
SetROP2	0x0104
SetStretchBltMode	0x0107
SetTextAlign	0x012E
SetTextCharExtra	0x0108
SetTextColor	0x0209
SetTextJustification	0x020A
SetViewportExt	0x020E

Table 9.2 GDI Functions and Values (*continued*)

Function	Value
SetViewportOrg	0x020D
SetWindowExt	0x020C
SetWindowOrg	0x020B

Typical Metafile Record

The following structured list shows the format of a typical metafile record:

```
struct{
 DWORD rdSize;
 WORD rdFunction;
 WORD rdParm[];
}
```

A typical metafile record contains the following fields:

Field	Description
rdSize	Specifies the size in words of the record.
rdFunction	Specifies the function number.
rdParm[ ]	Is an array of words containing the function parameters, in the reverse order in which they are passed to the function.

Function-Specific Records

Some metafile records contain data structures in the parameter field. This section contains definitions for these records.

AnimatePalette Record 3.0

The **AnimatePalette** record has the following format:

```
struct {
 DWORD rdSize;
 WORD rdFunction;
 WORD rdParm[];
}
```

This record contains the following fields:

Field	Description
rdSize	Specifies the record size in words.
rdFunction	Specifies the function number 0x0436.
rdParm[ ]	Contains the following elements:

Element	Description
start	First entry to be animated.
numentries	Number of entries to be animated.
entries	PALETTEENTRY blocks.

BitBlt Record (Prior to 3.0)

The **BitBlt** record stored by Windows versions prior to 3.0 contains a device-dependent bitmap which may not be suitable for playback on all devices. The following is the format of this record:

```
struct {
DWORD rdSize;
WORD rdFunction;
WORD rdParm[];
}
```

This record contains the following fields:

Field	Description
rdSize	Specifies the record size in words.
rdFunction	Specifies the function number 0x0922 .
rdParm[ ]	Contains the following elements:

Element	Description
raster op	High word of the raster operation.
SY	The *y*-coordinate of the source origin.
SX	The *x*-coordinate of the source origin.
DYE	Destination *y*-extent.

Field	Description	
	Element	**Description**
	DXE	Destination *x*-extent.
	DY	The *y*-coordinate of destination origin.
	DX	The *x*-coordinate of destination origin.
	bmWidth	Width of bitmap (in pixels)
	bmHeight	Height of bitmap (in raster lines)
	bmWidthBytes	Number of bytes in each raster line.
	bmPlanes	Number of color planes in the bitmap.
	bmBitsPixel	Number of adjacent color bits.
	bits	Actual device-dependent bitmap bits.

BitBlt Record 3.0

The **BitBlt** record stored by Windows versions 3.0 and later contains a device-independent bitmap suitable for playback on any device. The following is the format of this record:

```
struct {
DWORD rdSize;
WORD rdFunction;
WORD rdParm[];
}
```

This record contains the following fields:

Field	Description
rdSize	Specifies the record size in words.
rdFunction	Specifies the function number 0x0940.
rdParm[ ]	Contains the following elements:

Field	Description	
	Element	**Description**
	raster op	High word of the raster operation.
	SY	The *y*-coordinate of the source origin.
	SX	The *x*-coordinate of the source origin.
	DYE	The *y*-extent of the destination.
	DXE	The *x*-extent of the destination.
	DY	The *y*-coordinate of destination origin.
	DX	The *x*-coordinate of destination origin.
	BitmapInfo	**BITMAPINFO** data structure.
	bits	Actual device-independent bitmap bits.

CreateBrushIndirect Record

The **CreateBrushIndirect** record has the following format:

```
struct {
 DWORD rdSize;
 WORD rdFunction;
 LOGBRUSH rdParm;
}
```

This record contains the following fields:

Field	Description
rdSize	Specifies the record size in words.
rdFunction	Specifies the function number 0x02FC.
rdParm	Specifies the logical brush.

CreateFontIndirect Record

The **CreateFontIndirect** record has the following format:

```
struct {
 DWORD rdSize;
 WORD rdFunction;
 LOGFONT rdParm;
}
```

This record contains the following fields:

Field	Description
rdSize	Specifies the record size in words.
rdFunction	Specifies the function number 0x02FB.
rdParm	Specifies the logical font.

CreatePalette Record 3.0

The **CreatePalette** record has the following format:

```
struct {
 DWORD rdSize;
 WORD rdFunction;
 LOGPALETTE rdParm;
}
```

This record contains the following fields:

Field	Description
rdSize	Specifies the record size in words.
rdFunction	Specifies the function number 0x00F7.
rdParm	Specifies the logical palette.

CreatePatternBrush Record (Prior to 3.0)

The **CreatePatternBrush** record stored by Windows versions prior to 3.0 contains a device-dependent bitmap which may not be suitable for playback on all devices. The following is the format of this record:

```
struct {
 DWORD rdSize;
 WORD rdFunction;
 WORD rdParm[];
}
```

This record contains the following fields:

Field	Description
rdSize	Specifies the record size in words.
rdFunction	Specifies the function number 0x01F9.
rdParm[ ]	Contains the following elements:

Element	Description
bmWidth	Bitmap width.
bmHeight	Bitmap height.
bmWidthBytes	Bytes per raster line.
bmPlanes	Number of color planes.
bmBitsPixel	Number of adjacent color bits that define a pixel.
bmBits	Pointer to bit values.
bits	Actual bits of pattern.

CreatePatternBrush Record 3.0

The **CreatePatternBrush** record stored by Windows versions 3.0 and later contains a device-independent bitmap suitable for playback on all devices. The following is the format of this record:

```
struct {
 DWORD rdSize;
 WORD rdFunction;
 WORD rdParm[];
}
```

This record contains the following fields:

Field	Description
rdSize	Specifies the record size in words.
rdFunction	Specifies the function number 0x0142.
rdParm[ ]	Contains the following elements:

Field	Description	
	Element	**Description**
	type	Bitmap type. This field may be either of these two values: ■ BS_PATTERN—Brush is defined by a device-dependent bitmap through a call to the **CreatePatternBrush** function. ■ BS_DIBPATTERN—Brush is defined by a device-independent bitmap through a call to the **CreateDIBPatternBrush** function.
	Usage	Specifies whether the **bmiColors[ ]** field of the **BITMAPINFO** data structure contains explicit RGB values or indexes into the currently realized logical palette. This field must be one of the following values: ■ DIB_RGB_COLORS—The color table contains literal RGB values. ■ DIB_PAL_COLORS—The color table consists of an array of indexes into the currently realized logical palette.
	BitmapInfo	**BITMAPINFO** data structure.
	bits	Actual device-independent bitmap bits.

CreatePenIndirect Record

The **CreatePenIndirect** record has the following format:

```
struct {
 DWORD rdSize;
 WORD rdFunction;
 LOGPEN rdParm;
}
```

This record contains the following fields:

Field	Description
rdSize	Specifies the record size in words.
rdFunction	Specifies the function number 0x02FA.
rdParm	Specifies the logical pen.

Create Region Record

The **Create Region** record has the following format:

```
struct {
 DWORD rdSize;
 WORD rdFunction;
 WORD rdParm[];
}
```

This record contains the following fields:

Field	Description
rdSize	Specifies the record size in words.
rdFunction	Specifies the function number 0x06FF.
rdParm[ ]	Specifies the region to be created.

DeleteObject 3.0

The **DeleteObject** record has the following format:

```
struct {
 DWORD rdSize;
 WORD rdFunction;
 WORD rdParm;
}
```

This record contains the following fields:

Field	Description
rdSize	Specifies the record size in words.
rdFunction	Specifies the function number 0x01F0.
rdParm	Specifies the handle-table index of the object to be deleted.

DrawText Record

The **DrawText** record has the following format:

```
struct{
 DWORD rdSize;
 WORD rdFunction;
 WORD rdParm[];
}
```

This record contains the following fields:

Field	Description
rdSize	Specifies the record size in words.
rdFunction	Specifies the function number 0x062F.
rdParm[ ]	Contains the following elements:

Element	Description
format	Method of formatting.
count	Number of bytes in the string.
rectangle	Rectangular structure defining area where text is to be defined
string	Byte array containing the string. The array is ((count + 1) >> 1) words long.

Escape Record

The **Escape** record has the following format:

```
struct {
 DWORD rdSize;
 WORD rdFunction;
 WORD rdParm[];
}
```

This record contains the following fields:

Field	Description
rdSize	Specifies the record size in words.
rdFunction	Specifies the function number 0x0626.
rdParm[ ]	Contains the following elements:

Element	Description
escape number	Number identifying individual escape.
count	Number of bytes of information.
input data	Variable length field. The field is ((count+1)/ >> 1) words long.

ExtTextOut Record

The **ExtTextOut** record has the following format:

```
struct{
 DWORD rdSize;
 WORD rdFunction;
 WORD rdParm[];
}
```

This record contains the following fields:

Field	Description
rdSize	Specifies the record size in words.
rdFunction	Specifies the function number 0x0A32.
rdParm[ ]	Contains the following elements:

Element	Description
y	Logical *y*-value of string's starting point.
x	Logical *x*-value of string's starting point.

Field	Description	
	Element	**Description**
	count	Length of the string.
	options	Rectangle type.
	rectangle	**RECT** structure defining the **ExtTextOut** rectangle if options element is nonzero; nonexistent if options element equals zero.
	string	Byte array containing the string. The array is ((count + 1) >> 1) words long.
	dxarray	Optional word array of inter-character distances.

Polygon Record

The **Polygon** record has the following format:

```
struct {
 DWORD rdSize;
 WORD rdFunction;
 WORD rdParm[];
}
```

This record contains the following fields:

Field	Description	
rdSize	Specifies the record size in words.	
rdFunction	Specifies the function number 0x0324.	
rdParm[ ]	Contains the following elements:	
	Element	**Description**
	count	Number of points.
	list of points	List of individual points.

PolyPolygon Record

The **PolyPolygon** record has the following format:

```
struct {
 DWORD rdSize;
 WORD rdFunction;
 WORD rdParm[];
```

This record contains the following fields:

Field	Description
rdSize	Specifies the record size in words.
rdFunction	Specifies the function number 0x0538.
rdParm[ ]	Contains the following elements:

Element	Description
count	Total number of points.
list of polygon counts	List of number of points for each polygon.
list of points	List of individual points.

Polyline Record

The **Polyline** record has the following format:

```
struct {
 DWORD rdSize;
 WORD rdFunction;
 WORD rdParm[];
}
```

This record contains the following fields:

Field	Description
rdSize	Specifies the record size in words.
rdFunction	Specifies the function number 0x0325.
rdParm[ ]	Contains the following elements:

Field	Description	
	Element	**Description**
	count	Number of points.
	list of points	List of individual points.

SelectClipRegion

The **SelectClipRegion** record has the following format:

```
struct{
 DWORD rdSize;
 WORD rdFunction;
 WORD rdParm;
}
```

This record contains the following fields:

Field	Description
rdSize	Specifies the record size in words.
rdFunction	Specifies the function number 0x012C.
rdParm	Specifies the handle-table index of the region being selected.

SelectObject

The **SelectObject** record has the following format:

```
struct{
 DWORD rdSize;
 WORD rdFunction;
 WORD rdParm;
}
```

This record contains the following fields:

Field	Description
rdSize	Specifies the record size in words.
rdFunction	Specifies the function number 0x012D.
rdParm	Specifies the handle-table index of the object being selected.

SelectPalette Record 3.0

The **SelectPalette** record has the following format:

```
struct{
 DWORD rdSize;
 WORD rdFunction;
 WORD rdParm;
}
```

This record contains the following fields:

Field	Description
rdSize	Specifies the record size in words.
rdFunction	Specifies the function number 0x0234.
rdParm	Specifies the handle-table index of the logical palette being selected.

SetDIBitsToDevice Record 3.0

The **SetDIBitsToDevice** record has the following format:

```
struct {
 DWORD rdSize;
 WORD rdFunction;
 WORD rdParm[];
}
```

This record contains the following fields:

Field	Description
rdSize	Specifies the record size in words.
rdFunction	Specifies the function number 0x0D33.
rdParm[ ]	Contains the following elements:

Element	Description
wUsage	Flag indicating whether the bitmap color table contains RGB values or indexes into the currently realized logical palette.

Field	Description	
	Element	**Description**
	numscans	Number of scan lines in the bitmap.
	startscan	First scan line in the bitmap.
	srcY	The *y*-coordinate of the origin of the source in the bitmap.
	srcX	The *x*-coordinate of the origin of the source in the bitmap.
	extY	Height of the source in the bitmap.
	extX	Width of the source in the bitmap.
	destY	The *y*-coordinate of the origin of the destination rectangle.
	destX	The *x*-coordinate of the origin of the destination rectangle.
	BitmapInfo	**BITMAPINFO** data structure.
	bits	Actual device-independent bitmap bits.

SetPaletteEntries Record 3.0

The **SetPaletteEntries** record has the following format:

```
struct {
 DWORD rdSize;
 WORD rdFunction;
 WORD rdParm[];
}
```

This record contains the following fields:

Field	Description
rdSize	Specifies the record size in words.
rdFunction	Specifies the function number 0x0037.
rdParm[ ]	Contains the following elements:

Field	Description	
	Element	**Description**
	start	First entry to be set in the palette.
	numentries	Number of entries to be set in the palette.
	entries	PALETTEENTRY blocks.

StretchBlt Record (Prior to 3.0)

The **StretchBlt** record stored by Windows versions prior to 3.0 contains a device-dependent bitmap which may not be suitable for playback on all devices. The following is the format of this record:

```
struct {
 DWORD rdSize;
 WORD rdFunction;
 WORD rdParm[];
}
```

This record contains the following fields:

Field	Description	
rdSize	Specifies the record size in words.	
rdFunction	Specifies the function number 0x0B23.	
rdParm[ ]	Contains the following elements:	
	Element	**Description**
	raster op	Low word of the raster operation.
	raster op	High word of the raster operation.
	SYE	The *y*-extent of the source.
	SXE	The *x*-extent of the source.
	SY	The *y*-coordinate of the source origin.

Field	Description	
	Element	**Description**
	SX	The *x*-coordinate of the source origin.
	DYE	The *y*-extent of the destination.
	DXE	The *x*-extent of the destination.
	DY	The *y*-coordinate of destination origin.
	DX	The *x*-coordinate of destination origin.
	bmWidth	Width of the bitmap in pixels.
	bmHeight	Height of the bitmap in raster lines.
	bmWidthBytes	Number of bytes in each raster line.
	bmPlanes	Number of color planes in the bitmap.
	bmBitsPixel	Number of adjacent color bits.
	bits	Actual bitmap bits.

StretchBlt Record 3.0

The **StretchBlt** record stored by Windows versions 3.0 and later contains a device-independent bitmap suitable for playback on all devices. The following is the format of this record:

```
struct {
 DWORD rdSize;
 WORD rdFunction;
 WORD rdParm[];
}
```

This record contains the following fields:

Field	Description
rdSize	Specifies the record size in words.
rdFunction	Specifies the function number 0x0B41.
rdParm[ ]	Contains the following elements:

Element	Description
raster op	Low word of the raster operation.
raster op	High word of the raster operation.
SYE	The *y*-extent of the source.
SXE	The *x*-extent of the source.
SY	The *y*-coordinate of the source origin.
SX	The *x*-coordinate of the source origin.
DYE	The *y*-extent of the destination.
DXE	The *x*-extent of the destination.
DY	The *y*-coordinate of destination origin.
DX	The *x*-coordinate of destination origin.
BitmapInfo	**BITMAPINFO** data structure.
bits	Actual device-independent bitmap bits.

StretchDIBits Record 3.0

The **StretchDIBits** record has the following format:

```
struct {
 DWORD rdSize;
 WORD rdFunction;
 WORD rdParm[];
}
```

This record contains the following fields:

Field	Description
rdSize	Specifies the record size in words.
rdFunction	Specifies the function number 0x0F43.
rdParm[ ]	Contains the following elements:

Element	Description
dwRop	Raster operation to be performed.
wUsage	Flag indicating whether the bitmap color table contains RGB values or indexes into the currently realized logical palette.
srcYExt	Height of the source in the bitmap.
srcXExt	Width of the source in the bitmap.
srcY	The *y*-coordinate of the origin of the source in the bitmap.
srcX	The *x*-coordinate of the origin of the source in the bitmap.
dstYExt	Height of the destination rectangle.
dstXExt	Width of the destination rectangle.
dstY	The *y*-coordinate of the origin of the destination rectangle.
dstX	The *x*-coordinate of the origin of the destination rectangle.
BitmapInfo	**BITMAPINFO** data structure.
bits	Actual device-independent bitmap bits.

TextOut Record

The **TextOut** record has the following format:

```
struct {
 DWORD rdSize;
 WORD rdFunction;
 WORD rdParm[];
}
```

This record contains the following fields:

Field	Description
rdSize	Specifies the record size in words.
rdFunction	Specifies the function number 0x0521.
rdParm[ ]	Contains the following elements:

Element	Description
count	The string's length.
string	The actual string.
y-value	Logical *y*-coordinate of string's starting point.
x-value	Logical *x*-coordinate of string's starting point.

9.5.3 Sample Metafile Program Output

This section shows the metafile created by a sample program.

The following sample program creates a small metafile in which a purple rectangle with a green border is drawn, and the words "Hello People" are written in the rectangle.

```
MakeAMetaFile(hDC)
HDC hDC;
{
HPEN      hMetaGreenPen;
HBRUSH    hMetaVioletBrush;
HDC       hDCMeta;
HANDLE    hMeta;

/* create the metafile with output going to the disk
*/
hDCMeta = CreateMetaFile( (LPSTR) "sample.met");
```

```
hMetaGreenPen = CreatePen(0, 0, (DWORD) 0x0000FF00);
SelectObject(hDCMeta, hMetaGreenPen);

hMetaVioletBrush = CreateSolidBrush( (DWORD)
0x00FF00FF);
SelectObject(hDCMeta, hMetaVioletBrush);

Rectangle(hDCMeta, 0, 0, 150, 70);

TextOut(hDCMeta, 10, 10, (LPSTR) "Hello People", 12);

/* we are done with the metafile */
hMeta = CloseMetaFile(hDCMeta);

/* play the metafile that we just created */
PlayMetaFile(hDC, hMeta);
}
```

The resulting binary file SAMPLE.MET looks like this:

```
0001         mtType... disk metafile
0009         mtSize...
0100         mtVersion
0000 0036    mtSize
0002         mtNoObjects
0000 000C    mtMaxRecord
0000         mtNoParameters

0000 0008    rdSize
02FA         rdFunction (CreatePen function call)
0000 0000 0000 0000 FF00  rdParm (LOGPEN structure defining pen)

0000 0004    rdSize
012D         rdFunction (SelectObject)
0000         rdParm (index to object #0... the above pen)

0000 0007    rdSize
02FC         rdFunction (CreateBrush)
0000 00FF 00FF 0000 rdParm (LOGBRUSH structure defining the brush)

0000 0004    rdSize
012D         rdFunction (SelectObject)
0001         rdParm (index to object #1... the brush)

0000 0007    rdSize
041B         rdFunction (Rectangle)
0046 0096 0000 0000 rdParm (parameters sent to Rectangle...
                    in reverse order)

0000 000C    rdSize
0521         rdFunction (TextOut)
rdParm
```

```
000C          count
string
48 65 6C 6C 6F 20 50 65 6F 70 6C 65    "Hello People"
000A              y-value
000A              x-value
```

9.6 Summary

Windows files store information required to create Windows applications as well as data needed by the Windows system and Windows applications during execution. For more information on topics related to Windows files, see the following:

Topic	Reference
Metafile functions	*Reference, Volume 1*: Chapter 1, "Window Manager Interface Functions," and Chapter 4, "Functions Directory"
Device-independent bitmaps	*Guide to Programming*: Chapter 11, "Bitmaps"
Windows clipboard	*Guide to Programming*: Chapter 13, "The Clipboard"
Creating icons and cursors	*Tools*: Chapter 4, "Designing Images: SDKPaint"

Chapter 10

Module-Definition Statements

This chapter describes the statements contained in the module-definition file that defines the application's contents and system requirements for the **LINK** program. **LINK** links compiled source files with Microsoft Windows and other libraries to create an executable Windows application. For information on running **LINK**, see *Tools*.

The module-definition file contains one or more of the following module statements:

Statement	Description
CODE	Code-segment attributes
DATA	Data-segment attributes
DESCRIPTION	One-line description of the module
EXETYPE	.EXE header type (Windows or OS/2)
EXPORTS	Exported functions
HEAPSIZE	Size of local heap in bytes
IMPORTS	Imported functions
LIBRARY	Dynamic-link library name
NAME	Module name
SEGMENTS	Additional code segment
STACKSIZE	Size of local stack in bytes
STUB	Old-style executable

This chapter describes these statements, their syntax, required and optional parameters, and usage.

CODE

Syntax

CODE [[FIXED|MOVEABLE]] [[DISCARDABLE]] [[\PRELOAD|LOADONCALL]]

This statement defines the attributes of the standard code segment. The standard code segment is the application segment having the name _TEXT and belonging to the class CODE. In C applications, the standard data segment is created automatically if no specific segment name is included in the C-Compiler command line.

The **FIXED** option, if included, means that the segment remains at a fixed memory location; the **MOVEABLE** option means that the segment can be moved, if necessary, in order to compact memory.

The **DISCARDABLE** option, if included, means that the segment can be discarded if it is no longer needed.

The **PRELOAD** option, if included, means that the segment is loaded when the module is first loaded; the **LOADONCALL** option means that the segment is loaded when it is called. The Resource Compiler may override this option. See *Tools* for more information.

Comments

There are no default attributes for code segments. The .DEF file should always explicitly define code-segment attributes.

If conflicting options are included in the same statement, **LINK** uses the overriding option to determine the segment attributes. The following list shows which options override which:

MOVEABLE overrides **FIXED.**

PRELOAD overrides **LOADONCALL.**

Example

```
CODE MOVEABLE LOADONCALL
```

In this example, the loader forces all fixed and moveable (but not discardable) data segments to be loaded. Libraries cannot have code that is moveable but not discardable.

DATA

Syntax

DATA [[NONE|SINGLE|MULTIPLE]] [[FIXED|MOVEABLE]]

This statement defines the attributes of the standard data segment. The standard data segment is all application segments belonging to the group DGROUP and the class DATA. In C applications, the standard data segment is created automatically. The data is always preloaded.

The **NONE** option, if included, means that there is no data segment. To be effective, this option should be the only attribute of the segment. This option is available only for libraries.

The **SINGLE** option, if included, means that a single segment is shared by all instances of the module, and is valid only for libraries.

The **MULTIPLE** option means that one segment exists for each instance, and is only valid for applications.

NONE, **SINGLE**, and **MULTIPLE** are mutually exclusive.

The **FIXED** option, if included, means that the segment remains at a fixed memory location. The **MOVEABLE** option means that the segment can be moved if necessary, in order to compact memory.

Comments

There are no default attributes for data segments. The .DEF file should always explicitly define data-segment attributes.

Data segments are always preloaded.

If conflicting options are included in the same statement, **LINK** uses the overriding option to determine the segment attributes. The following list shows which options override which:

MULTIPLE overrides **NONE.**

SINGLE overrides **NONE.**

MOVEABLE overrides **FIXED.**

Example

```
DATA MOVEABLE SINGLE
```

This example tells **LINK** that this module has a single, moveable data segment.

DESCRIPTION

Syntax

DESCRIPTION '*text*'

This statement inserts text into the application's module. It is useful for embedding source-control or copyright information.

Parameter	Description
text	Specifies one or more ASCII characters. The string must be enclosed in single quotation marks.

Example

```
DESCRIPTION 'Microsoft Windows Template Application'
```

This example embeds the text "Microsoft Windows Template Application" in the application module.

EXETYPE

Syntax

EXETYPE *headertype*

This statement specifies the default executable-file (.EXE) header type (Windows or OS/2). It is required for every Windows application.

Parameter	Description
headertype	Determines the header type. When linking an application intended for the Windows environment, you must set this parameter to the value "WINDOWS". For an MS OS/2 application, set this parameter to the value "OS/2".

Example

```
EXETYPE WINDOWS
```

EXPORTS

Syntax

EXPORTS *exportname* [[*ordinal-option*]] [[*res-option*]] [[*data-option*]] [[*parameter-option*]]

This statement defines the names and attributes of the functions to be exported to other applications. The **EXPORTS** key word marks the beginning of the definitions. It can be followed by any number of export definitions, each on a separate line.

Parameter	Description
exportname	Specifies one or more ASCII characters that define the function name. It has the following form: <*entryname*>=[[*internalname*]] where the *entryname* parameter specifies the name to be used by other applications to access the exported function, and *internalname* is an optional parameter that defines the actual name of the function if *entryname* is not the actual name.
ordinal-option	Defines the function's ordinal value. It has the following form:

Parameter	Description
	@ordinal where *ordinal* takes an integer value that specifies the function's ordinal value. The ordinal value defines the location of the function's name in the application's string table. (When exporting functions from libraries, it is better to use an ordinal rather than a name; using ordinals conserves space.)
res-option	Is the optional key word **RESIDENTNAME**, which specifies that the function's name must be resident at all times.
data-option	Is the optional key word **NODATA**, which specifies that the function is not bound to a specific data segment. When invoked, the function uses the current data segment.
parameter-option	Is an optional integer value that specifies the number of words the function expects to be passed as parameters.

Example

```
EXPORTS
        SampleRead=read2bin @1 8
        StringIn=str1 @2 4
        CharTest NODATA
```

This example exports the functions SampleRead, StringIn and CharTest so that other applications, or Windows itself, can call them.

HEAPSIZE

Syntax

HEAPSIZE *bytes*

This statement defines the number of bytes needed by the application for its local heap. An application uses the local heap whenever it allocates local memory.

The default heap size is zero. The minimum size is 256 bytes. For an application, the size of the local heap must be at least large enough to hold the current environment.

Parameter	Description
bytes	Is an integer value that specifies the heap size in bytes. It must not exceed 65,536 (the size of a single physical segment).

Example

```
HEAPSIZE 4096
```

This example sets the size of the application's local heap to 4096 bytes.

IMPORTS

Syntax

IMPORTS [[*internal-option*]] *modulename* [[*entry-option*]]

This statement defines the names and attributes of the functions to be imported from dynamic-link libraries. The **IMPORTS** key word marks the beginning of the definitions. It can be followed by any number of import definitions, each on a separate line.

Parameter	Description
internal-option	Specifies the name that the application will use to call the function. It has the following form: *internal-name=* where *internal-name* is one or more ASCII characters. This name must be unique.
modulename	Specifies one or more uppercase ASCII characters that define the name of the executable module that contains the function. The module name must match the name of the executable file. For example, an application with the executable file SAMPLE.DLL has the module name "SAMPLE". The executable file must be named with the .DLL extension.
entry-option	Specifies the function to be imported. It can be one of the following: *.entryname* *.entryordinal* where *entryname* is the actual name of the function, and *entryordinal* is the ordinal value of the function.

Example

```
IMPORTS
        Sample.SampleRead
        write2hex=Sample.SampleWrite
        Read.1
```

NOTE Instead of listing imported DLL functions in the **IMPORTS** statement, you can specify an "import library" for the DLL in your application's **LINK** command line. It also saves space to import by ordinal.

LIBRARY

Syntax

LIBRARY *libraryname*

This statement defines the name of a library module. Library modules are resource modules that contain code, data, and other resources but are not intended to be executed as an independent program. Like an application's module name, a library's module name must match the name of the executable file. For example, the library USER.EXE has the module name "USER".

Parameter	Description
libraryname	Specifies one or more ASCII characters that define the name of the library module.

Comments

The start address of the library module is determined by the library's object files; it is an internally defined function.

The *libraryname* parameter is optional. If the parameter is not included, **LINK** uses the filename part of the executable file (that is, the name with the extension removed).

If the .DEF file includes neither a **NAME** nor a **LIBRARY** statement, **LINK** assumes a **NAME** statement without a *modulename* parameter is desired.

Example

```
LIBRARY Utilities
```

This example gives a library the module name "Utilities."

NAME

Syntax

NAME *modulename*

This statement defines the name of the application's executable module. The module name identifies the module when exporting functions.

Parameter	Description
modulename	Specifies one or more uppercase ASCII characters that define the name of the executable module. The module name must match the name of the executable file. For example, an application with the executable file SAMPLE.EXE has the module name "SAMPLE".

Comments

The *modulename* parameter is optional. If the parameter is not included, **LINK** assumes that the module name matches the the filename of the executable file. For example, if you do not specify a module name and the executable file is named MYAPP.EXE, **LINK** assumes that the module name is "MYAPP".

If the .DEF file includes neither a **NAME** nor a **LIBRARY** statement, **LINK** assumes a **NAME** statement without a *modulename* parameter is desired.

Example

```
NAME Calendar
```

This example gives an application the module name "Calendar".

SEGMENTS

Syntax

SEGMENTS *segmentname* [[**CLASS** '*class-name*']] [[*minalloc*]]\
[[**FIXED|MOVEABLE**]]
[[**DISCARDABLE**]] [[**SHARED|NONSHARED**]] [[**PRELOAD|LOADONCALL**]]

This statement defines the segment attributes of additional code and data segments.

The **FIXED** option, if included, means that the segment remains at a fixed memory location. The **MOVEABLE** option means that the segment can be moved if necessary, in order to compact memory.

The **DISCARDABLE** option, if included, means that the segment can be discarded if it is no longer needed.

The **PRELOAD** option, if included, means that the segment is loaded immediately The **LOADONCALL** option means that the segment is loaded when it is accessed or called. The Resource Compiler may override this option. See *Tools* for more information.

Parameter	Description
segmentname	Specifies a character string that names the new segment. It can be any name, including the standard segment names _TEXT and _DATA, which represent the standard code and data segments.

Parameter	Description
class-name	Is an optional key word that specifies the class name of the specified segment. If no class name is specified, **LINK** uses the class name CODE by default.
minalloc	Is an optional integer value that specifies the minimum allocation size for the segment.

Comments

There are no default attributes for additional segments. The .DEF file should always explicitly define the attributes of additional segments.

If conflicting options are included in the same statement, **LINK** uses the overriding option to determine the segment attributes. The following list shows which options override which:

MOVEABLE overrides **FIXED**.

PRELOAD overrides **LOADONCALL**.

Example

```
SEGMENTS
     _TEXT FIXED
     _INIT PRELOAD DISCARDABLE
     _RES  CLASS 'DATA' PRELOAD DISCARDABLE
```

STACKSIZE

Syntax

STACKSIZE *bytes*

This statement defines the number of bytes needed by the application for its local stack. An application uses the local stack whenever it makes function calls.

The default stack size is zero if the application makes no function calls. If your application does make function calls and you specify a stack size smaller than 5K bytes, Windows automatically sets the stack size to 5K bytes.

Parameter	Description
bytes	Is an integer value that specifies the stack size in bytes.

Comments

Do not use the **STACKSIZE** statement for dynamic-link libraries.

Example

```
STACKSIZE 6144
```

This example sets the size of an application's stack to 6144 bytes.

STUB

Syntax

STUB '*filename*'

This statement appends the old-style executable file specified by *filename* to the beginning of the module. The executable stub should display a warning message and terminate if the user attempts to execute the module without having loaded Windows. The default file WINSTUB.EXE can be used if no other actions are required.

Parameter	Description
filename	Specifies the name of the old-style executable file that will be appended to the module. The name must have the DOS filename format.

Comments

If the file named by *filename* is not in the current directory, **LINK** searches for the file in the directories specified by the user's PATH environment variable.

Example

```
STUB 'WINSTUB.EXE'
```

This example specifies the executable file WINSTUB.EXE as the application's stub. If a user tries to run this application in the DOS environment, rather than with Windows, the program WINSTUB.EXE starts instead.

Chapter 11 Binary and Ternary Raster-Operation Codes

This chapter lists and describes the binary and ternary raster operations used by the graphics device interface (GDI). A binary raster operation uses two operands: a pen and a destination bitmap. A ternary raster operation uses three operands: a source bitmap, a brush, and a destination bitmap. Both binary and ternary raster operations use Boolean operators.

11.1 Binary Raster Operations

This section lists the binary raster-operation codes used by the **GetROP2** and **SetROP2** functions. Raster-operation codes define how GDI combines the bits from the selected pen with the bits in the destination bitmap.

Each raster-operation code represents a Boolean operation in which the selected pen and the destination bitmap are combined. There are two operands used in these operations:

D	Destination bitmap
P	Selected pen

The Boolean operators used in these operations are as follows:

a	Bitwise AND
n	Bitwise NOT (inverse)
o	Bitwise OR
x	Bitwise Exclusive OR (XOR)

All Boolean operations are presented in reverse Polish notation. For example, the following operation replaces the destination with a combination of the pen and the selected brush:

```
DPo
```

Each raster-operation code is a 32-bit integer value whose high-order word is a Boolean operation index and whose low-order word is the operation code. The 16-bit operation index is a zero-extended 8-bit value that represents the result of the Boolean operation on predefined pen and destination values. For example, the operation indexes for the DPo and DPan operations are shown in Table 11.1:

Table 11.1 Operation Indexes for DPo and DPan

P	D	PSo	DPSoo
0	0	0	1
0	1	1	1
1	0	1	1
1	1	1	0

The following list outlines the drawing modes and the Boolean operations that they represent:

Raster Operation	Boolean Operation
R2_BLACK	0
R2_COPYPEN	P
R2_MASKNOTPEN	DPna
R2_MASKPEN	DPa
R2_MASKPENNOT	PDna
R2_MERGENOTPEN	DPno
R2_MERGEPEN	DPo
R2_MERGEPENNOT	PDno
R2_NOP	D
R2_NOT	Dn
R2_NOTCOPYPEN	Pn
R2_NOTMASKPEN	DPan
R2_NOTMERGEPEN	DPon
R2_NOTXORPEN	DPxn
R2_WHITE	1
R2_XORPEN	DPx

When a monochrome device is used, GDI maps the value zero to black and the value 1 to white. Given an application that attempts to draw with a black pen on a white destination by using the available binary raster operations, the following results will occur:

Raster Operation	Result
R2_BLACK	Visible black line
R2_COPYPEN	Visible black line
R2_MASKNOTPEN	No visible line
R2_MASKPEN	Visible black line
R2_MASKPENNOT	Visible black line
R2_MERGENOTPEN	No visible line
R2_MERGEPEN	Visible black line
R2_MERGEPENNOT	Visible black line
R2_NOP	No visible line
R2_NOT	Visible black line
R2_NOTCOPYPEN	No visible line
R2_NOTMASKPEN	No visible line
R2_NOTMERGEPEN	Visible black line
R2_NOTXORPEN	Visible black line
R2_WHITE	No visible line
R2_XORPEN	No visible line

When a color device is used, GDI uses RGB values to represent the colors of the pen and the destination. An RGB color value is a long integer that contains a red, a green, and a blue color field, each specifying the intensity of the given color. Intensities range from 0 to 255. The values are packed in the three low-order bytes of the long integer. The color of a pen is always a solid color, but the color of the destination may be a mixture of any two or three colors. Given an application that attempts to draw with a white pen on a blue destination by using the available binary raster operations, the following results will occur:

Raster Operation	Result
R2_BLACK	Visible black line
R2_COPYPEN	Visible white line
R2_MASKNOTPEN	Visible black line

Raster Operation	Result
R2_MASKPEN	Invisible blue line
R2_MASKPENNOT	Visible red/green line
R2_MERGENOTPEN	Invisible blue line
R2_MERGEPEN	Visible white line
R2_MERGEPENNOT	Visible white line
R2_NOP	Invisible blue line
R2_NOT	Visible red/green line
R2_NOTCOPYPEN	Visible black line
R2_NOTMASKPEN	Visible red/green line
R2_NOTMERGEPEN	Visible black line
R2_NOTXORPEN	Invisible blue line
R2_WHITE	Visible white line
R2_XORPEN	Visible red/green line

11.2 Ternary Raster Operations

This section lists the ternary raster-operation codes used by the **BitBlt**, **PatBlt**, and **StretchBlt** functions. Ternary raster-operation codes define how GDI combines the bits in a source bitmap with the bits in the destination bitmap.

Each raster-operation code represents a Boolean operation in which the source, the selected brush, and the destination bitmap are combined. There are three operands used in these operations:

D	Destination bitmap
P	Selected brush (also called pattern)
S	Source bitmap

The Boolean operators used in these operations are as follows:

a	Bitwise AND
n	Bitwise NOT (inverse)
o	Bitwise OR
x	Bitwise Exclusive OR (XOR)

All Boolean operations are presented in reverse Polish notation. For example, the following operation replaces the destination with a combination of the source and brush:

```
PSo
```

The following operation combines the source and brush with the destination (there are alternate spellings of the same function, so although a particular spelling may not be in the list, an equivalent form will be):

```
DPSoo
```

Each raster-operation code is a 32-bit integer value whose high-order word is a Boolean operation index and whose low-order word is the operation code. The 16-bit operation index is a zero-extended, 8-bit value that represents the result of the Boolean operation on predefined brush, source, and destination values. For example, the operation indexes for the PSo and DPSoo operations are shown in Table 11.2:

Table 11.2 Operation Indexes for PSo and DPSoo

P	S	D	PSo	DPSoo
0	0	0	0	0
0	0	1	0	1
0	1	0	1	1
0	1	1	1	1
1	0	0	1	1
1	0	1	1	1
1	1	0	1	1
1	1	1	1	1
Operation index:			00FC	00FE

In this case, PSo has the operation index 00FC (read from the bottom up); DPSoo has the operation index 00FE. These values define the location of the corresponding raster-operation codes, as shown in Table 11.1, "Operation Indexes for DPo and DPan." The PSo operation is in line 252 (FCh) of the table; DPSoo is in line 254 (FEh).

The most commonly used raster operations have been given special names in the Windows include file, WINDOWS.H. You should use these names whenever possible in your applications.

When the source and destination are monochrome, a bit value of zero represents a black pixel and a bit value of 1 represents a white pixel. When the source and the destination are color, those colors are represented with RGB values. For more information about RGB values, see the **RGB** structure in Chapter 7, "Data Types and Structures."

Table 11.3 lists the raster-operation codes:

Table 11.3 Raster-Operation Codes

Boolean Function in Hex	Hex ROP	Boolean Function in Reverse Polish	Common Name
00	00000042	0	BLACKNESS
01	00010289	DPSoon	-
02	00020C89	DPSona	-
03	000300AA	PSon	-
04	00040C88	SDPona	-
05	000500A9	DPon	-
06	00060865	PDSxnon	-
07	000702C5	PDSaon	-
08	00080F08	SDPnaa	-
09	00090245	PDSxon	-
0A	000A0329	DPna	-
0B	000B0B2A	PSDnaon	-
0C	000C0324	SPna	-
0D	000D0B25	PDSnaon	-
0E	000E08A5	PDSonon	-
0F	000F0001	Pn	-
10	00100C85	PDSona	-
11	001100A6	DSon	NOTSRCERASE
12	00120868	SDPxnon	-
13	001302C8	SDPaon	-
14	00140869	DPSxnon	-
15	001502C9	DPSaon	-
16	00165CCA	PSDPSanaxx	-
17	00171D54	SSPxDSxaxn	-
18	00180D59	SPxPDxa	-
19	00191CC8	SDPSanaxn	-
1A	001A06C5	PDSPaox	-
1B	001B0768	SDPSxaxn	-
1C	001C06CA	PSDPaox	-

Table 11.3 Raster-Operation Codes *(continued)*

Boolean Function in Hex	Hex ROP	Boolean Function in Reverse Polish	Common Name
1D	001D0766	DSPDxaxn	-
1E	001E01A5	PDSox	-
1F	001F0385	PDSoan	-
20	00200F09	DPSnaa	-
21	00210248	SDPxon	-
22	00220326	DSna	-
23	00230B24	SPDnaon	-
24	00240D55	SPxDSxa	-
25	00251CC5	PDSPanaxn	-
26	002606C8	SDPSaox	-
27	00271868	SDPSxnox	-
28	00280369	DPSxa	-
29	002916CA	PSDPSaoxxn	-
2A	002A0CC9	DPSana	-
2B	002B1D58	SSPxPDxaxn	-
2C	002C0784	SPDSoax	-
2D	002D060A	PSDnox	-
2E	002E064A	PSDPxox	-
2F	002F0E2A	PSDnoan	-
30	0030032A	PSna	-
31	00310B28	SDPnaon	-
32	00320688	SDPSoox	-
33	00330008	Sn	NOTSRCCOPY
34	003406C4	SPDSaox	-
35	00351864	SPDSxnox	-
36	003601A8	SDPox	-
37	00370388	SDPoan	-
38	0038078A	PSDPoax	-
39	00390604	SPDnox	-
3A	003A0644	SPDSxox	-
3B	003B0E24	SPDnoan	-
3C	003C004A	PSx	-
3D	003D18A4	SPDSonox	-

Table 11.3 Raster-Operation Codes *(continued)*

Boolean Function in Hex	Hex ROP	Boolean Function in Reverse Polish	Common Name
3E	003E1B24	SPDSnaox	-
3F	003F00EA	PSan	-
40	00400F0A	PSDnaa	-
41	00410249	DPSxon	-
42	00420D5D	SDxPDxa	-
43	00431CC4	SPDSanaxn	-
44	00440328	SDna	SRCERASE
45	00450B29	DPSnaon	-
46	004606C6	DSPDaox	-
47	0047076A	PSDPxaxn	-
48	00480368	SDPxa	-
49	004916C5	PDSPDaoxxn	-
4A	004A0789	DPSDoax	-
4B	004B0605	PDSnox	-
4C	004C0CC8	SDPana	-
4D	004D1954	SSPxDSxoxn	-
4E	004E0645	PDSPxox	-
4F	004F0E25	PDSnoan	-
50	00500325	PDna	-
51	00510B26	DSPnaon	-
52	005206C9	DPSDaox	-
53	00530764	SPDSxaxn	-
54	005408A9	DPSonon	-
55	00550009	Dn	DSTINVERT
56	005601A9	DPSox	-
57	00570389	DPSoan	-
58	00580785	PDSPoax	-
59	00590609	DPSnox	-
5A	005A0049	DPx	PATINVERT
5B	005B18A9	DPSDonox	-
5C	005C0649	DPSDxox	-
5D	005D0E29	DPSnoan	-
5E	005E1B29	DPSDnaox	-

Table 11.3 Raster-Operation Codes *(continued)*

Boolean Function in Hex	Hex ROP	Boolean Function in Reverse Polish	Common Name
5F	005F00E9	DPan	-
60	00600365	PDSxa	-
61	006116C6	DSPDSaoxxn	-
62	00620786	DSPDoax	-
63	00630608	SDPnox	-
64	00640788	SDPSoax	-
65	00650606	DSPnox	-
66	00660046	DSx	SRCINVERT
67	006718A8	SDPSonox	-
68	006858A6	DSPDSonoxxn	-
69	00690145	PDSxxn	-
6A	006A01E9	DPSax	-
6B	006B178A	PSDPSoaxxn	-
6C	006C01E8	SDPax	-
6D	006D1785	PDSPDoaxxn	-
6E	006E1E28	SDPSnoax	-
6F	006F0C65	PDSxnan	-
70	00700CC5	PDSana	-
71	00711D5C	SSDxPDxaxn	-
72	00720648	SDPSxox	-
73	00730E28	SDPnoan	-
74	00740646	DSPDxox	-
75	00750E26	DSPnoan	-
76	00761B28	SDPSnaox	-
77	007700E6	DSan	-
78	007801E5	PDSax	-
79	00791786	DSPDSoaxxn	-
7A	007A1E29	DPSDnoax	-
7B	007B0C68	SDPxnan	-
7C	007C1E24	SPDSnoax	-
7D	007D0C69	DPSxnan	-
7E	007E0955	SPxDSxo	-
7F	007F03C9	DPSaan	-

Table 11.3 Raster-Operation Codes *(continued)*

Boolean Function in Hex	Hex ROP	Boolean Function in Reverse Polish	Common Name
80	008003E9	DPSaa	-
81	00810975	SPxDSxon	-
82	00820C49	DPSxna	-
83	00831E04	SPDSnoaxn	-
84	00840C48	SDPxna	-
85	00851E05	PDSPnoaxn	-
86	008617A6	DSPDSoaxx	-
87	008701C5	PDSaxn	-
88	008800C6	DSa	SRCAND
89	00891B08	SDPSnaoxn	-
8A	008A0E06	DSPnoa	-
8B	008B0666	DSPDxoxn	-
8C	008C0E08	SDPnoa	-
8D	008D0668	SDPSxoxn	-
8E	008E1D7C	SSDxPDxax	-
8F	008F0CE5	PDSanan	-
90	00900C45	PDSxna	-
91	00911E08	SDPSnoaxn	-
92	009217A9	DPSDPoaxx	-
93	009301C4	SPDaxn	-
94	009417AA	PSDPSoaxx	-
95	009501C9	DPSaxn	-
96	00960169	DPSxx	-
97	0097588A	PSDPSonoxx	-
98	00981888	SDPSonoxn	-
99	00990066	DSxn	-
9A	009A0709	DPSnax	-
9B	009B07A8	SDPSoaxn	-
9C	009C0704	SPDnax	-
9D	009D07A6	DSPDoaxn	-
9E	009E16E6	DSPDSaoxx	-
9F	009F0345	PDSxan	-
A0	00A000C9	DPa	-

Table 11.3 Raster-Operation Codes *(continued)*

Boolean Function in Hex	Hex ROP	Boolean Function in Reverse Polish	Common Name
A1	00A11B05	PDSPnaoxn	-
A2	00A20E09	DPSnoa	-
A3	00A30669	DPSDxoxn	-
A4	00A41885	PDSPonoxn	-
A5	00A50065	PDxn	-
A6	00A60706	DSPnax	-
A7	00A707A5	PDSPoaxn	-
A8	00A803A9	DPSoa	-
A9	00A90189	DPSoxn	-
AA	00AA0029	D	-
AB	00AB0889	DPSono	-
AC	00AC0744	SPDSxax	-
AD	00AD06E9	DPSDaoxn	-
AE	00AE0B06	DSPnao	-
AF	00AF0229	DPno	-
B0	00B00E05	PDSnoa	-
B1	00B10665	PDSPxoxn	-
B2	00B21974	SSPxDSxox	-
B3	00B30CE8	SDPanan	-
B4	00B4070A	PSDnax	-
B5	00B507A9	DPSDoaxn	-
B6	00B616E9	DPSDPaoxx	-
B7	00B70348	SDPxan	-
B8	00B8074A	PSDPxax	-
B9	00B906E6	DSPDaoxn	-
BA	00BA0B09	DPSnao	-
BB	00BB0226	DSno	MERGEPAINT
BC	00BC1CE4	SPDSanax	-
BD	00BD0D7D	SDxPDxan	-
BE	00BE0269	DPSxo	-
BF	00BF08C9	DPSano	-
C0	00C000CA	PSa	MERGECOPY
C1	00C11B04	SPDSnaoxn	-

Table 11.3 Raster-Operation Codes (*continued*)

Boolean Function in Hex	Hex ROP	Boolean Function in Reverse Polish	Common Name
C2	00C21884	SPDSonoxn	-
C3	00C3006A	PSxn	-
C4	00C40E04	SPDnoa	-
C5	00C50664	SPDSxoxn	-
C6	00C60708	SDPnax	-
C7	00C707AA	PSDPoaxn	-
C8	00C803A8	SDPoa	-
C9	00C90184	SPDoxn	-
CA	00CA0749	DPSDxax	-
CB	00CB06E4	SPDSaoxn	-
CC	00CC0020	S	SRCCOPY
CD	00CD0888	SDPono	-
CE	00CE0B08	SDPnao	-
CF	00CF0224	SPno	-
D0	00D00E0A	PSDnoa	-
D1	00D1066A	PSDPxoxn	-
D2	00D20705	PDSnax	-
D3	00D307A4	SPDSoaxn	-
D4	00D41D78	SSPxPDxax	-
D5	00D50CE9	DPSanan	-
D6	00D616EA	PSDPSaoxx	-
D7	00D70349	DPSxan	-
D8	00D80745	PDSPxax	-
D9	00D906E8	SDPSaoxn	-
DA	00DA1CE9	DPSDanax	-
DB	00DB0D75	SPxDSxan	-
DC	00DC0B04	SPDnao	-
DD	00DD0228	SDno	-
DE	00DE0268	SDPxo	-
DF	00DF08C8	SDPano	-
E0	00E003A5	PDSoa	-
E1	00E10185	PDSoxn	-
E2	00E20746	DSPDxax	-

Table 11.3 Raster-Operation Codes *(continued)*

Boolean Function in Hex	Hex ROP	Boolean Function in Reverse Polish	Common Name
E3	00E306EA	PSDPaoxn	-
E4	00E40748	SDPSxax	-
E5	00E506E5	PDSPaoxn	-
E6	00E61CE8	SDPSanax	-
E7	00E70D79	SPxPDxan	-
E8	00E81D74	SSPxDSxax	-
E9	00E95CE6	DSPDSanaxxn	-
EA	00EA02E9	DPSao	-
EB	00EB0849	DPSxno	-
EC	00EC02E8	SDPao	-
ED	00ED0848	SDPxno	-
EE	00EE0086	DSo	SRCPAINT
EF	00EF0A08	SDPnoo	-
F0	00F00021	P	PATCOPY
F1	00F10885	PDSono	-
F2	00F20B05	PDSnao	-
F3	00F3022A	PSno	-
F4	00F40B0A	PSDnao	-
F5	00F50225	PDno	-
F6	00F60265	PDSxo	-
F7	00F708C5	PDSano	-
F8	00F802E5	PDSao	-
F9	00F90845	PDSxno	-
FA	00FA0089	DPo	-
FB	00FB0A09	DPSnoo	PATPAINT
FC	00FC008A	PSo	-
FD	00FD0A0A	PSDnoo	-
FE	00FE02A9	DPSoo	-
FF	00FF0062	1	WHITENESS

11.3 Summary

Raster-operation codes define how GDI combines the bits of a source bitmap with the bits of a destination bitmap. For more information on topics related to raster-operation codes, see the following:

Topic	Reference
Using raster-operation codes with GDI functions	*Reference, Volume 1*: Chapter 2, "Graphics Device Interface Functions," and Chapter 4, "Functions Directory"
Setting the current drawing mode with **SetROP2**	*Reference, Volume 1*: Chapter 4, "Functions Directory"
	Guide to Programming: Chapter 6, "The Cursor, the Mouse, and the Keyboard"
Bitmaps and raster operations	*Guide to Programming*: Chapter 11, "Bitmaps"

Chapter 12 Printer Escapes

This chapter contains an alphabetical list of the individual Microsoft Windows printer escapes. The printer escapes allow applications to access facilities of a particular output device that are not available directly through the graphics device interface (GDI). The escape calls are made by an application, translated by Windows, and then sent to the printer device driver.

ABORTDOC

Syntax

short Escape(*hDC*, ABORTDOC, NULL, NULL, NULL)

This escape terminates the current job, erasing everything the application has written to the device since the last **ENDDOC** escape.

The **ABORTDOC** escape should be used to terminate:

- Printing operations that do not specify an abort function using the **SETABORTPROC** escape
- Printing operations that have not yet reached their first **NEWFRAME** or **NEXTBAND** escape call

Parameter	Type/Description
hDC	**HDC** Identifies the device context.

Return Value

The return value specifies the outcome of the escape. It is positive if the escape is successful. Otherwise, it is negative.

Comments

If an application encounters a printing error or a canceled print operation, it must not attempt to terminate the operation by using the **Escape** function with either the **ENDDOC** or **ABORTDOC** escape. GDI automatically terminates the operation before returning the error value.

If the application displays a dialog box to allow the user to cancel the print operation, it must send the **ABORTDOC** escape before destroying the dialog box.

The application must send the **ABORTDOC** escape before freeing the procedure-instance address of the abort function, if any.

BANDINFO

Syntax

short Escape(*hDC*, BANDINFO, sizeof(BANDINFOSTRUCT), *lpInData*, *lpOutData*)

This escape copies information about a device with banding capabilities to a structure pointed to by the *lpOutData* parameter. It is implemented only for devices that use banding.

Banding is a property of an output device that allows a page of output to be stored in a metafile and divided into bands, each of which is sent to the device to create a complete page.

The information copied to the structure pointed to by *lpOutData* includes:

- A value that indicates whether there are graphics in the next band
- A value that indicates whether there is text on the page
- A **RECT** data structure that contains a bounding rectangle for all graphics on the page

The *lpOutData* parameter is NULL if no data are returned.

The *lpInData* parameter specifies information sent by the application to the device driver. This information is read by the device driver only on the first **BANDINFO** escape call on a page.

Parameter	Type/Description
hDC	**HDC** Identifies the device context.
lpInData	**BANDINFOSTRUCT FAR *** Points to a **BANDINFOSTRUCT** data structure that contains information to be passed to the driver. See the following "Comments" section for more information on the **BANDINFOSTRUCT** data structure.
lpOutData	**BANDINFOSTRUCT FAR *** Points to a **BANDINFOSTRUCT** data structure that contains information returned by the driver. See the following "Comments" section for more information on the **BANDINFOSTRUCT** data structure.

Return Value

The return value specifies the outcome of the escape. It is 1 if the escape is successful. It is zero if the function fails or is not implemented by the driver.

Comments

The **BANDINFOSTRUCT** data structure contains information about the contents of a page and supplies a bounding rectangle for graphics on the page. The following shows the format of **BANDINFOSTRUCT**:

```
typedef struct {
    BOOL   fGraphicsFlag;
    BOOL   fTextFlag;
    RECT   GraphicsRect;
} BANDINFOSTRUCT;
```

The **BANDINFOSTRUCT** structure has the following fields:

Field	Description
fGraphicsFlag	Is TRUE if graphics are or are expected to be on the page or in the band; otherwise, it is FALSE.
fTextFlag	Is TRUE if text is or is expected to be on the page or in the band; otherwise, it is FALSE.
GraphicsRect	Contains a **RECT** data structure that supplies a bounding region for all graphics on the page.

Table 12.1 shows the meaning of these fields, depending on which parameter contains the structure.

Table 12.1 Meaning of BANDINFOSTRUCT Fields

Field	When Used in *lpInData*	When Used in *lpOutData*
fGraphicsFlag	TRUE if the application is informing the driver that graphics are on the page.	TRUE if the driver is informing the application that it expects graphics in this band.
fTextFlag	TRUE if the application is informing the driver that text is on the page.	TRUE if the driver is informing the application that it expects text in this band.
GraphicsRect	Supplies the bounding rectangle for all graphics on the page.	No valid return data.

An application should call this escape immediately after each call to the **NEXTBAND** escape. It is in reference to the band the driver returned to that escape.

An application should use this escape in the following manner:

On the first band, the driver may give the application a full-page band and ask for text only (**fGraphicsFlag** is set to FALSE and **fTextFlag** is set to TRUE). The application sends only text to the driver.

If in the first band the application indicated that it had graphics (**fGraphicsFlag** is set to TRUE), or that the driver encountered vector fonts, then the driver will band the rest of the page. If there are no graphics or vector fonts, then the next **NEXTBAND** will return an empty rectangle to indicate that the application should move on to the next page.

If there are graphics but no vector fonts (the application set **fGraphicsFlag** to TRUE, but there were no graphics in the first full-page text band), then for subsequent bands the driver may optionally band only into the rectangle the application passed. This rectangle bounds all graphics on the page. If there are vector fonts, then the driver will band the entire width and depth of the page with **fTextFlag** set to TRUE. It will also set **fGraphicsFlag** to true if the application set it.

The driver assumes that an application using **BANDINFO** will only send text in the first full-page text band since that is all the driver requested. Therefore, if the driver encounters a vector font or graphics in the band, it assumes they were generated by a text primitive and sets **fTextFlag** to TRUE for all subsequent graphics bands so they can be output as graphics. If the application does not satisfy this expectation, the image will still be generated properly, but the driver will spend time sending spurious text primitives to graphics bands.

Older drivers written before the **BANDINFO** escape was designed used full-page banding for text. If a particular driver does not support the **BANDINFO** escape but sets RC_BANDING, the application can detect full-page banding for text by determining if the first band on the page covers the entire page.

BEGIN_PATH

Syntax

short Escape(*hDC*, BEGIN_PATH, NULL, NULL, NULL)

This escape opens a path. A path is a connected sequence of primitives drawn in succession to form a single polyline or polygon. Paths enable applications to draw complex borders, filled shapes, and clipping areas by supplying a collection of other primitives that define the desired shape.

Printer escapes supporting paths enable applications to render images on sophisticated devices such as PostScript® printers without generating huge polygons to simulate the images.

To draw a path, an application first issues the **BEGIN_PATH** escape. It then draws the primitives defining the border of the desired shape and issues an **END_PATH** escape. The **END_PATH** escape includes a parameter specifying how the path is to be rendered.

Parameter	Type/Description
hDC	**HDC** Identifies the device context.

Return Value

The return value specifies the current path nesting level. If the escape is successful, the return value is the number of **BEGIN_PATH** escape calls without a corresponding **END_PATH** escape call. Otherwise, the return value is zero.

Comments

An application may begin a subpath within another path. If the subpath is closed, it is treated exactly like a polygon. If it is open, it is treated exactly like a polyline.

An application may use the **CLIP_TO_PATH** escape to define a clipping area corresponding to the interior or exterior of the currently open path.

CLIP_TO_PATH

Syntax

short Escape(*hDC*, CLIP_TO_PATH, sizeof(int), *lpClipMode*, NULL)

This escape defines a clipping area bounded by the currently open path. It enables the application to save and restore the current clipping area and to set up an inclusive or exclusive clipping area bounded by the currently open path. If the path defines an inclusive clipping area, portions of primitives falling outside the interior bounded by the path are clipped. If the path defines an exclusive clipping area, portions of primitives falling inside the interior are clipped.

Parameter	Type/Description
hDC	**HDC** Identifies the device context.
lpClipMode	**LPINT** Points to a short integer specifying the clipping mode. It can be one of the following values:

Value	Meaning
CLIP_SAVE (0)	Saves the current clipping area.
CLIP_RESTORE (1)	Restores the previous clipping area.
CLIP_INCLUSIVE (2)	Sets an inclusive clipping area.
CLIP_EXCLUSIVE (3)	Sets an exclusive clipping area.

Return Value

The return value specifies the outcome of the escape. It is nonzero if the escape was successful. Otherwise, it is zero.

Comments

To clip a set of primitives against a path, an application should follow these steps:

1. Save the current clipping area using the CLIP_TO_PATH escape.
2. Begin a path using the BEGIN_PATH escape.
3. Draw the primitives bounding the clipping area.

4. Set the clipping area using the CLIP_TO_PATH escape.
5. Close the path using the END_PATH escape.
6. Draw the primitives to be clipped.
7. Restore the original clipping area using the CLIP_TO_PATH escape.

DEVICEDATA

Syntax **short Escape(*hDC*, DEVICEDATA,** *nCount*, *lpInData*, *lpOutData*)

This escape is identical to the **PASSTHROUGH** escape. See the description of **PASSTHROUGH** for further information.

DRAFTMODE

Syntax **short Escape(*hDC*, DRAFTMODE, sizeof(int),** *lpDraftMode*, **NULL)**

This escape turns draft mode off or on. Turning draft mode on instructs the device driver to print faster and with lower quality (if necessary). The draft mode can be changed only at page boundaries, for example, after a **NEWFRAME** escape directing the driver to advance to a new page.

Parameter	Type/Description
hDC	**HDC** Identifies the device context.
lpDraftMode	**LPINT** Points to a short-integer value that specifies the draft mode. It may be one of the following values:

Value	Meaning
0	Specifies draft mode off.
1	Specifies draft mode on.

Return Value The return value specifies the outcome of the escape. It is positive if the escape is successful. Otherwise, it is negative.

Comments The default draft mode is off.

DRAWPATTERNRECT

Syntax

short Escape(*hDC*, DRAWPATTERNRECT, sizeof(PRECTSTRUCT), ***lpInData*****,** **NULL)**

This escape creates a pattern, gray scale, or solid black rectangle by using the pattern/rule capabilities of Page Control Language (PCL) on Hewlett-Packard® LaserJet® or LaserJet-compatible printers. A gray scale is a gray pattern that contains a specific mixture of black and white pixels.

Parameter	Type/Description
hDC	**HDC** Identifies the device context.
lpInData	**PRECT_STRUCT FAR *** Points to a **PRECT_STRUCT** data structure that describes the rectangle. See the following "Comments" section for more information on the **PRECT_STRUCT** data structure.

Return Value

The return value specifies the outcome of the escape. It is 1 if the escape is successful. Otherwise, it is zero.

Comments

The *lpInData* parameter points to a **PRECT_STRUCT** data structure that defines the rectangle to be created. The **PRECT_STRUCT** structure has the following format:

```
typedef struct {
    POINT  prPosition;
    POINT  prSize;
    WORD   prStyle;
    WORD   prPattern;
} PRECT_STRUCT;
```

This structure has the following fields:

Field	Description
prPosition	Specifies the upper-left corner of the rectangle.
prSize	Specifies the lower-right corner of the rectangle.
prStyle	Specifies the type of pattern. It may be one of the following values:

Field	Description	
	Value	**Meaning**
	0	Black rule.
	1	White rule that erases bitmap data previously written to same area; this pattern is available on the HP LaserJet IIP only.
	2	Gray scale.
	3	HP-defined.
prPattern	Specifies the pattern. It is ignored for a black rule. It specifies the percentage of gray for a gray-scale pattern. It represents one of six Hewlett-Packard-defined patterns.	

An application should use the **QUERYESCSUPPORT** escape to determine whether a device is capable of drawing patterns and rules before using the **DRAWPATTERNRECT** escape. If an application uses the **BANDINFO** escape, all patterns and rectangles sent by using **DRAWPATTERNRECT** should be treated as text and sent on a text band.

Do not try to erase patterns and rules created with the **DRAWPATTERNRECT** escape by placing opaque objects over them. To erase such patterns and rules, use the function calls provided by GDI.

ENABLEDUPLEX

Syntax **short Escape(*hDC*, ENABLEDUPLEX, sizeof(WORD), *lpInData*, NULL)**

This escape enables the duplex printing capabilities of a printer. A device that possesses duplex printing capabilities is able to print on both sides of the output medium.

Parameter	Type/Description
hDC	**HDC** Identifies the device context.
lpInData	**WORD FAR *** Points to an unsigned 16-bit integer that specifies whether duplex or simplex printing is used. It may be one of the following values:

Parameter	Type/Description	
	Value	**Meaning**
	0	Simplex
	1	Duplex with vertical binding
	2	Duplex with horizontal binding

Return Value The return value specifies the outcome of the escape. It is 1 if the escape is successful. Otherwise, it is zero.

Comments An application should use the **QUERYESCSUPPORT** escape to determine whether an output device is capable of creating duplex output. If **QUERYESCSUPPORT** returns a nonzero value, the application should send the **ENABLEDUPLEX** escape even if simplex printing is desired. This guarantees replacement of any values set in the driver-specific dialog box. If duplex printing is enabled and an uneven number of **NEXTFRAME** escapes are sent to the driver prior to the **ENDDOC** escape, the driver will eject an additional page before ending the print job.

ENABLEPAIRKERNING

Syntax **short Escape(***hDC*, **ENABLEPAIRKERNING, sizeof(int),** *lpNewKernFlag*, *lpOldKernFlag*)

This escape enables or disables the driver's ability to kern character pairs automatically. Kerning is the process of adding or subtracting space between characters in a string of text.

When pair kerning is enabled, the driver automatically kerns those pairs of characters that are listed in the font's character-pair kerning table. The driver reflects this kerning both on the printer and in **GetTextExtent** function calls.

Parameter	Type/Description
hDC	**HDC** Identifies the device context.
lpNewKernFlag	**LPINT** Points to a short-integer value that specifies whether automatic pair kerning is to be enabled (1) or disabled (0).
lpOldKernFlag	**LPINT** Points to a short-integer value that will receive the previous automatic pair-kerning value.

Return Value

The return value specifies the outcome of the escape. It is 1 if the escape is successful; it is zero if the escape is not successful or not implemented.

Comments

The default state of this escape is zero; automatic character-pair kerning is disabled.

A driver does not have to support the **ENABLEPAIRKERNING** escape just because it supplies the character-pair kerning table to the application via the **GETPAIRKERN-TABLE** escape. In the case where the **GETPAIRKERNTABLE** escape is supported but the **ENABLEPAIRKERNING** escape is not, the application must properly space the kerned characters on the output device using the **ExtTextOut** function.

ENABLERELATIVEWIDTHS

Syntax

short Escape(*hDC*, **ENABLERELATIVEWIDTHS, sizeof(int),** *lpNewWidthFlag*, *lpOldWidthFlag*)

This escape enables or disables relative character widths. When relative widths are disabled (the default), each character's width can be expressed as a number of device units. This guarantees that the extent of a string will equal the sum of the extents of the characters in the string. This allows applications to build an extent table by using one-character **GetTextExtent** function calls.

When relative widths are enabled, the sum of a string may not equal the sum of the widths of the characters. Applications that enable this feature are expected to retrieve the font's extent table and compute relatively scaled string widths.

Parameter	Type/Description
hDC	**HDC** Identifies the device context.
lpNewWidthFlag	**LPINT** Points to a short-integer value that specifies whether relative widths are to be enabled (1) or disabled (0).
lpOldWidthFlag	**LPINT** Points to a short-integer value that will receive the previous relative character width value.

Return Value

The return value specifies the outcome of the escape. It is 1 if the escape is successful; it is zero if the escape is not successful or not implemented.

Comments

The default state of this escape is zero; relative character widths are disabled.

The values specified as font units and accepted and returned by the escapes described in this chapter are returned in the relative units of the font when the **ENABLERELATIVE-WIDTHS** escape is enabled.

It is assumed that only linear-scaling devices will be dealt with in a relative mode. Non-linear-scaling devices do not implement this escape.

ENDDOC

Syntax

short Escape(*hDC*, ENDDOC, NULL, NULL, NULL)

This escape ends a print job started by a **STARTDOC** escape.

Parameter	Type/Description
hDC	**HDC** Identifies the device context.

Return Value

The return value specifies the outcome of the escape. It is positive if the escape is successful. Otherwise, it is negative.

Comments

If an application encounters a printing error or a canceled print operation, it must not attempt to terminate the operation by using the **Escape** function with either the **ENDDOC** or **ABORTDOC** escape. GDI automatically terminates the operation before returning the error value.

If the application displays a dialog box to allow the user to cancel the print operation, it must send the **ENDDOC** escape before destroying the dialog box.

The application must send the **ENDDOC** escape before freeing the procedure-instance address of the abort function, if any.

END_PATH

Syntax

short Escape(*hDC*, END_PATH, sizeof(PATH_INFO), *lpInData*, NULL)

This escape ends a path. A path is a connected sequence of primitives drawn in succession to form a single polyline or polygon. Paths enable applications to draw complex borders, filled shapes, and clipping areas by supplying a collection of other primitives defining the desired shape.

Printer escapes supporting paths enable applications to render images on sophisticated devices such as PostScript printers without generating huge polygons to simulate them.

To draw a path, an application first issues the **BEGIN_PATH** escape. It then draws the primitives defining the border of the desired shape and issues an **END_PATH** escape.

The **END_PATH** escape takes as a parameter a pointer to a structure specifying the manner in which the path is to be rendered. The structure specifies whether or not the path is to be drawn and whether it is open or closed. Open paths define polylines, and closed paths define fillable polygons.

Parameter	Type/Description
hDC	**HDC** Identifies the device context.
lpInData	**PATH_INFO FAR *** Points to a **PATH_INFO** data structure that defines how the path is to be rendered. See the following "Comments" section for more information on this data structure.

Return Value

The return value specifies the current path nesting level. If the escape is successful, the return value is the number of **BEGIN_PATH** escape calls without a corresponding **END_PATH** call. Otherwise, the return value is –1.

Comments

An application may begin a subpath within another path. If the subpath is closed, it is treated exactly like a polygon. If it is open, it is treated exactly like a polyline.

An application may use the **CLIP_TO_PATH** escape to define a clipping area corresponding to the interior or exterior of the currently open path.

The *lpInData* parameter points to a **PATH_INFO** data structure that specifies how to render the path. This data structure has the following form:

```
typedef struct {
      short    RenderMode;
      BYTE     FillMode;
      BYTE     BkMode;
      LOGPEN   Pen;
      LOGBRUSH Brush;
      DWORD    BkColor;
}PATH_INFO;
```

The **PATH_INFO** structure has the following fields:

Field	Description
RenderMode	Specifies how the path is to be rendered. It may be one of the following values: **Value** / **Meaning** NO_DISPLAY (0) — The path is not drawn. OPEN (1) — The path is drawn as an open polygon. CLOSED (2) — The path is drawn as a closed polygon.
FillMode	Specifies how the path is to be filled. It can be one of the following values: **Value** / **Meaning** ALTERNATE (1) — The fill is done using the alternate fill algorithm. WINDING (2) — The fill is done using the winding fill algorithm.
BkMode	Specifies the background mode for filling the path. It can be one of the following values: **Value** / **Meaning** OPAQUE — The background is filled with the background color before the brush is drawn. TRANSPARENT — The background is not changed.
Pen	Specifies the pen with which the path is to be drawn. If **RenderMode** is set to NO_DISPLAY, the pen is ignored.
Brush	Specifies the brush with which the path is to be filled. If **RenderMode** is set to NO_DISPLAY or OPEN, the brush is ignored.
BkColor	Specifies the color with which the path is filled if **BkMode** is set to OPAQUE.

ENUMPAPERBINS

Syntax

short Escape(*hDC*, ENUMPAPERBINS, sizeof(int), *lpNumBins*, *lpOutData***)**

This escape retrieves attribute information about a specified number of paper bins. The **GETSETPAPERBINS** escape retrieves the number of bins available on a printer.

Parameter	Type/Description
hDC	**HDC** Identifies the device context.
lpNumBins	**LPINT** Points to an integer that specifies the number of bins for which information is to be retrieved.
lpOutData	**LPSTR** Points to a data structure to which information about the paper bins is copied. The size of the structure depends on the number of bins for which information was requested. See the following "Comments" section for a description of this data structure.

Return Value

The return value specifies the outcome of the escape. It is 1 if the escape is successful; it is zero if the escape is not successful or not implemented.

Comments

The data structure to which the *lpOutData* parameter points consists of two arrays. The first is an array of short integers containing the paper-bin identifier numbers in the following format:

```
short     BinList[cBinMax]
```

The number of integers in the array (*cBinMax*) is equal to the value pointed to by the *lpNumBins* parameter.

The second array in the data structure to which *lpOutData* points is an array of characters in the following format:

```
char PaperNames[cBinMax][cchBinName]
```

The *cBinMax* value is equal to the value pointed to by the *lpNumBins* parameter; the *cchBinName* value is the length of each string (currently 24).

ENUMPAPERMETRICS

Syntax

short Escape(*hDC*, ENUMPAPERMETRICS, sizeof(int), *lpMode*, *lpOutData***)**

This escape performs one of two functions according to the mode:

- It determines the number of paper types supported and returns this value, which can then be used to allocate an array of **RECT** data structures.
- It returns one or more **RECT** data structures that define the areas on the page that can receive an image.

Parameter	Type/Description
hDC	**HDC** Identifies the device context.
lpMode	**LPINT** Points to an integer that specifies the mode for the escape. It can be one of the following values:

Value	Meaning
0	The return value indicates how many **RECT** data structures are required to contain the information about the available paper types.
1	The array of **RECT** structures to which *lpOutData* points is filled with the information.

Parameter	Type/Description
lpOutData	**LPRECT** Points to an array of **RECT** data structures that return all the areas that can receive an image.

Return Value

The return value is positive if successful, zero if the escape is not implemented, and negative if an error occurred.

EPSPRINTING

Syntax

short Escape(*hDC*, EPSPRINTING, sizeof(BOOL), *lpBool*, **NULL)**

This escape suppresses the output of the Windows PostScript header control section, which is about 7K. If an application uses this escape, no GDI calls are allowed.

Parameter	Type/Description
hDC	**HDC** Identifies the device context.
lpBool	**BOOL FAR *** Points to a Boolean value indicating that downloading should be enabled (TRUE) or disabled (FALSE).

Return Value The return value is positive if successful, zero if the escape is not implemented, and negative if an error occurred.

EXT_DEVICE_CAPS

Syntax **short Escape(***hDC*, **EXT_DEVICE_CAPS, sizeof(int),** *lpIndex*, *lpCaps***)**

This escape retrieves information about device-specific capabilities. It supplements the **GetDeviceCaps** function.

Parameter	Type/Description
hDC	**HDC** Identifies the device context.
lpIndex	**LPINT** Points to a short integer specifying the index of the capability to be retrieved. It can be any one of the following values:

Value	Meaning
R2_CAPS (1)	The *lpCaps* parameter indicates which of the 16 binary raster operations the device driver supports. A bit will be set for each supported raster operation. For further information, see the description of the **SetROP2** function in Chapter 4, "Functions Directory," in *Reference, Volume 1.*

Parameter	Type/Description	
	Value	**Meaning**
	PATTERN_CAPS (2)	The *lpCaps* parameter returns the maximum dimensions of a pattern brush bitmap. The low-order word of the capability value contains the maximum width of a pattern brush bitmap, and the high-order word contains the maximum height.
	PATH_CAPS (3)	The *lpCaps* parameter indicates whether the device is capable of creating paths using alternate and winding interiors, and whether the device can do exclusive or inclusive clipping to path interiors. The path capabilities are obtained using the logical OR operation on the following values: PATH_ALTERNATE (1) PATH_WINDING (2) PATH_INCLUSIVE (4) PATH_EXCLUSIVE (8)
	POLYGON_CAPS (4)	The *lpCaps* parameter returns the maximum number of polygon points supported by the device. The capability value is an unsigned value specifying the maximum number of points.
	PATTERN_COLOR_CAPS (5)	The *lpCaps* parameter indicates whether the device can convert monochrome pattern bitmaps to color. The capability value is 1 if the device can do pattern bitmap color conversions, and zero if it cannot.

Parameter	Type/Description	
	Value	**Meaning**
	R2_TEXT_CAPS (6)	The *lpCaps* parameter indicates whether the device is capable of performing binary raster operations on text. The low-order word of the capability value specifies which raster operations are supported for text. A bit is set for each supported raster operation, as in the R2_CAPS escape. The high-order word specifies the type of text to which the raster capabilities apply. It is obtained by applying the logical OR operation to the following values together: RASTER_TEXT (1) DEVICE_TEXT (2) VECTOR_TEXT (4)
lpCaps	**DWORD FAR *** Points to a 32-bit integer to which the capabilities will be copied.	

Return Value

The return value is nonzero if the specified extended capability is supported, and zero if it is not.

EXTTEXTOUT

Syntax

short Escape(*hDC***, EXTTEXTOUT, sizeof(EXTTEXT_STRUCT),** *lpInData***, NULL)**

This escape provides an efficient way for the application to call the GDI **TextOut** function when justification, letter spacing, and/or kerning are involved.

This function is provided only for backward compatibility. New applications should use the GDI **ExtTextOut** function instead.

Parameter	Type/Description
hDC	**HDC** Identifies the device context.
lpInData	**EXTTEXT_STRUCT FAR *** Points to an **EXTTEXT_STRUCT** data structure that specifies the initial position, characters, and character widths of the string. See the following "Comments" section for more information on the **EXTTEXT_STRUCT** data structure.

Return Value The return value specifies the outcome of the escape. It is 1 if the escape is successful; it is zero if the escape is not successful or not implemented.

Comments The **EXTEXT_STRUCT** data structure has the following format:

```
typedef struct {
    WORD      X;
    WORD      Y;
    WORD FAR *lpText;
    WORD FAR *lpWidths;
} EXTTEXT_STRUCT;
```

This structure has the following fields.

Field	Description
X	Specifies the *x*-coordinate of the upper-left corner of the string's starting point.
Y	Specifies the *y*-coordinate of the upper-left corner of the string's starting point.
lpText	Points to an array of *cch* character codes, where *cch* is the number of bytes in the string (*cch* is also the number of words in the width array).
lpWidths	Points to an array of *cch* character widths to use when printing the string. The first character appears at (**X,Y**), the second at (**X** + **lpWidths**[0],**Y**), the third at (**X** + **lpWidths**[0] + **lpWidths**[1],**Y**), and so on. These character widths are specified in the font units of the currently selected font. (The character widths will always be equal to device units unless the application has enabled relative character widths.) The units contained in the width array are specified as font units of the device.

FLUSHOUTPUT

Syntax **short Escape(*hDC*, FLUSHOUTPUT, NULL, NULL, NULL)**

This escape clears all output from the device's buffer.

Parameter	Type/Description
hDC	**HDC** Identifies the device context.

Return Value The return value specifies the outcome of the escape. It is positive if the escape is successful. Otherwise, it is negative.

GETCOLORTABLE

Syntax **short Escape(*hDC*, GETCOLORTABLE, sizeof(int),** *lpIndex*, *lpColor*)

This escape retrieves an RGB color-table entry and copies it to the location specified by the *lpColor* parameter.

Parameter	Type/Description
hDC	**HDC** Identifies the device context.
lpIndex	**LPINT** Points to a short-integer value that specifies the index of a color-table entry. Color-table indexes start at zero for the first table entry.
lpColor	**DWORD FAR** * Points to the long-integer value that will receive the RGB color value for the given entry.

Return Value The return value specifies the outcome of the escape. It is positive if the escape is successful. Otherwise, it is negative.

GETEXTENDEDTEXTMETRICS

Syntax **short Escape(*hDC*, GETEXTENDEDTEXTMETRICS, sizeof(WORD),** *lpInData*, *lpOutData*)

This escape fills the buffer pointed to by the *lpOutData* parameter with the extended text metrics for the selected font.

Parameter	Type/Description
hDC	**HDC** Identifies the device context.
lpInData	**WORD FAR *** Points to an unsigned 16-bit integer that specifies the number of bytes pointed to by the *lpOutData* parameter.
lpOutData	**EXTTEXTMETRIC FAR *** Points to an **EXTTEXTMETRIC** data structure. See the following "Comments" section for a description of this data structure.

Return Value

The return value specifies the number of bytes copied to the buffer pointed to by the *lpOutData* parameter. This value will never exceed that specified in the *nSize* field pointed to by the *lpInData* parameter. The return value is zero if the escape fails or is not implemented.

Comments

The *lpOutData* parameter points to an **EXTTEXTMETRIC** data structure which has the following format:

```
typedef struc{
   short etmSize;
   short etmPointSize;
   short etmOrientation;
   short etmMasterHeight;
   short etmMinScale;
   short etmMaxScale;
   short etmMasterUnits;
   short etmCapHeight;
   short etmXHeight;
   short etmLowerCaseAscent;
   short etmLowerCaseDescent;
   short etmSlant;
   short etmSuperScript;
   short etmSubScript;
   short etmSuperScriptSize;
   short etmSubScriptSize;
   short etmUnderlineOffset;
   short etmUnderlineWidth;
   short etmDoubleUpperUnderlineOffset;
   short etmDoubleLowerUnderlineOffset;
   short etmDoubleUpperUnderlineWidth;
   short etmDoubleLowerUnderlineWidth;
   short etmStrikeOutOffset;
   short etmStrikeOutWidth;
   WORD  etmKernPairs;
   WORD  etmKernTracks;
   }EXTTEXTMETRIC;
```

The **EXTTEXTMETRIC** data structure has the following fields:

Field	Description
etmSize	Specifies the size of the structure in bytes.
etmPointSize	Specifies the nominal point size of this font in twips (twentieths of a point, or 1/1440 inch). This is the intended size of the font; the actual size may differ slightly depending on the resolution of the device.
etmOrientation	Specifies the orientation of the font. The **etmOrientation** field may be any of the following values:

Value	Meaning
0	Either orientation
1	Portrait
2	Landscape

These values refer to the ability of this font to be placed on a page with the given orientation. A portrait page has a height that is greater than its width. A landscape page has a width that is greater than its height.

Field	Description
etmMasterHeight	Specifies the font size in device units for which the values in this font's extent table are exact.
etmMinScale	Specifies the minimum valid size for this font. The following equation illustrates how the minimum point size is determined:

$$\text{smallest point size} = \frac{\mathit{etmMinScale} \times 72}{\mathit{dfVertRes}}$$

The value 72 represents the number of points per inch. The *dfVertRes* value is the number of dots per inch.

Field	Description
etmMaxScale	Specifies the maximum valid size for this font. The following equation illustrates how the maximum point size is determined:

$$\text{largest point size} = \frac{\mathit{etmMaxScale} \times 72}{\mathit{dfVertRes}}$$

Field	Description
	The value 72 represents the number of points per inch. The *dfVertRes* value is the number of dots per inch.
etmMasterUnits	Specifies the integer number of units per em where an em equals **etmMasterHeight**. That is, **etmMasterUnits** is **emtMasterHeight** expressed in font units rather than device units.
etmCapHeight	Specifies the height in font units of uppercase characters in the font. Typically, this is the height of the capital H.
etmXHeight	Specifies the height in font units of lowercase characters in the font. Typically, this is the height of the lowercase x.
etmLowerCaseAscent	Specifies the distance in font units that the ascender of lowercase letters extends above the baseline. Typically, this is the height of the lowercase d.
etmLowerCaseDescent	Specifies the distance in font units that the descender of lowercase letters extends below the baseline. Typically, this is specified for the descender of the lowercase p.
etmSlant	Specifies for an italicized or slanted font the angle of the slant measured in tenths of a degree clockwise from the upright version of the font.
etmSuperScript	Specifies in font units the recommended amount to offset superscript characters from the baseline. This is typically a negative value.
etmSubScript	Specifies in font units the recommended amount to offset subscript characters from the baseline. This is typically a positive value.
etmSuperScriptSize	Specifies in font units the recommended size of superscript characters for this font.

Field	Description
etmSubScriptSize	Specifies in font units the recommended size of subscript characters for this font.
etmUnderlineOffset	Specifies in font units the offset downward from the baseline where the top of a single underline bar should appear.
etmUnderlineWidth	Specifies in font units the thickness of the underline bar.
etmDoubleUpperUnderlineOffset	Specifies the offset in font units downward from the baseline where the top of the upper double underline bar should appear.
etmDoubleLowerUnderlineOffset	Specifies the offset in font units downward from the baseline where the top of the lower double underline bar should appear.
etmDoubleUpperUnderlineWidth	Specifies in font units the thickness of the upper underline bar.
etmDoubleLowerUnderlineWidth	Specifies in font units the thickness of the lower underline bar.
etmStrikeOutOffset	Specifies in font units the offset upward from the baseline where the top of a strike-out bar should appear.
etmStrikeOutWidth	Specifies the thickness in font units of the strike-out bar.
etmKernPairs	Specifies the number of character kerning pairs defined for this font. An application can use this value to calculate the size of the pair-kern table returned by the **GETPAIRKERNTABLE** escape. It will not be greater than 512 kern pairs.
etmKernTracks	Specifies the number of kerning tracks defined for this font. An application can use this value to calculate the size of the track-kern table returned by the **GETTRACKKERNTABLE** escape. It will not be greater than 16 kern tracks.

The values returned in many of the fields of the **EXTTEXTMETRIC** structure are affected by whether relative character widths are enabled or disabled. For more information, see the description of **ENABLERELATIVEWIDTHS** escape earlier in this chapter.

GETEXTENTTABLE

Syntax

short Escape(*hDC***, GETEXTENTTABLE, sizeof(CHAR_RANGE_STRUCT),** *lpInData*, *lpOutData***)**

This escape retrieves the width (extent) of individual characters from a group of consecutive characters in the selected font's character set.

Parameter	Type/Description
hDC	**HDC** Identifies the device context.
lpInData	**LPSTR** Points to a **CHAR_RANGE_STRUCT** data structure that defines the range of characters for which the width is to be retrieved. See the following "Comments" section for more information on the **CHAR_RANGE_STRUCT** data structure.
lpOutData	**LPINT** Points to an array of short integers that receives the character widths. The size of the array must be at least (**chLast** – **chFirst** + 1).

Return Value

The return value specifies the outcome of the escape. It is 1 if the escape is successful, and zero if the escape is not successful. If the escape is not implemented, the return value is zero.

Comments

The *lpInData* parameter points to a **CHAR_RANGE_STRUCT** data structure that defines the range of characters for which the width is to be retrieved. The **CHAR_RANGE_STRUCT** structure has the following format:

```
typedef struct {
    BYTE chFirst;
    BYTE chLast;
} CHAR_RANGE_STRUCT
```

This structure has the following fields:

Field	Description
chFirst	Specifies the character code of the first character whose width is to be retrieved.
chLast	Specifies the character code of the last character whose width is to be retrieved.

The values retrieved are affected by whether relative character widths are enabled or disabled. For more information, see the **ENABLERELATIVEWIDTHS** escape, earlier in this chapter.

GETFACENAME

Syntax **short Escape(***hDC***, GETFACENAME, NULL, NULL,** *lpFaceName***)**

This escape retrieves the face name of the current physical font.

Parameter	Type/Description
hDC	**HDC** Identifies the device context.
lpFaceName	**LPSTR** Points to a buffer of characters to receive the face name. This buffer must be at least 60 bytes in length.

Return Value The return value is positive if the escape was successful, zero if the escape is not implemented, or negative if an error occurred.

GETPAIRKERNTABLE

Syntax **short Escape(***hDC***, GETPAIRKERNTABLE, NULL, NULL,** *lpOutData***)**

This escape fills the buffer pointed to by the *lpOutData* parameter with the character-pair kerning table for the selected font.

Parameter	Type/Description
hDC	**HDC** Identifies the device context.
lpOutData	**KERNPAIR FAR *** Points to an array of **KERNPAIR** data structures. This array must be large enough to accommodate the font's entire character-pair kerning table. The number of character-kerning pairs in the font can be obtained from the **EXTTEXTMETRIC** data structure returned by the **GETEXTENDEDTEXTMETRICS** escape. See the following "Comments" section for the format of the **KERNPAIR** data structure.

Return Value

The return value specifies the number of **KERNPAIR** structures copied to the buffer. This value is zero if the font does not have kerning pairs defined, the escape fails, or is not implemented.

Comments

The **KERNPAIR** data structure has the following format:

```
typedef struc {
   union {
           BYTE each  [2];  /*  UNION:  'each'  and 'both'
                                 share the same memory */
           WORD both;
           } kpPair;
   short     kpKernAmount;
   } KERNPAIR;
```

The **KERNPAIR** structure contains the following fields:

Field	Description
kpPair.each[0]	Specifies the character code for the first character in the kerning pair.
kpPair.each[1]	Specifies the character code for the second character in the kerning pair.
kpPair.both	Specifies a **WORD** in which the first character in the kerning pair is in the low-order byte and the second character is in the high-order byte.
kpKernAmount	Specifies the signed amount that this pair will be kerned if they appear side by side in the same font and size. This value is typically negative since pair-kerning usually results in two characters being set more tightly than normal.

The array of **KERNPAIR** structures is sorted in increasing order by the **kpPair.both** field.

The values returned in the **KERNPAIR** structures are affected by whether relative character widths are enabled or disabled. For more information, see the description of the **ENABLERELATIVEWIDTHS** escape earlier in this chapter.

GETPHYSPAGESIZE

Syntax short **Escape(***hDC*, **GETPHYSPAGESIZE, NULL, NULL,** *lpDimensions***)**

This escape retrieves the physical page size and copies it to the location pointed to by the *lpDimensions* parameter.

Parameter	Type/Description
hDC	**HDC** Identifies the device context.
lpDimensions	**LPPOINT** Points to a **POINT** data structure that will receive the physical page dimensions. The **x** field of the **POINT** data structure receives the horizontal size in device units, and the **y** field receives the vertical size in device units.

Return Value The return value specifies the outcome of the escape. It is positive if the escape is successful. Otherwise, it is negative.

GETPRINTINGOFFSET

Syntax short **Escape(***hDC*, **GETPRINTINGOFFSET, NULL, NULL,** *lpOffset***)**

This escape retrieves the offset from the upper-left corner of the physical page where the actual printing or drawing begins. This escape is generally not useful for devices that allow the user to set the origin of the printable area directly.

Parameter	Type/Description
hDC	**HDC** Identifies the device context.
lpOffset	**LPPOINT** Points to a **POINT** structure that will receive the printing offset. The **x** field of the **POINT** structure receives the horizontal coordinate of the printing offset in device units, and the **y** field receives the vertical coordinate of the printing offset in device units.

Return Value The return value specifies the outcome of the escape. It is positive if the escape is successful. Otherwise, it is negative.

GETSCALINGFACTOR

Syntax **short Escape(***hDC***, GETSCALINGFACTOR, NULL, NULL,** *lpFactors***)**

This escape retrieves the scaling factors for the *x*- and *y*-axes of a printing device. For each scaling factor, the escape copies an exponent of 2 to the location pointed to by the *lpFactors* parameter. For example, the value 3 is copied to *lpFactors* if the scaling factor is 8.

Scaling factors are used by printing devices that support graphics at a smaller resolution than text.

Parameter	Type/Description
hDC	**HDC** Identifies the device context.
lpFactors	**LPPOINT** Points to the **POINT** data structure that will receive the scaling factor. The **x** field of the **POINT** structure receives the scaling factor for the *x*-axis, and the **y** field receives the scaling factor for the *y*-axis.

Return Value The return value specifies the outcome of the escape. It is positive if the escape is successful. Otherwise, it is negative.

GETSETPAPERBINS

Syntax **short Escape(***hDC***, GETSETPAPERBINS,** *nCount*, *lpInData*, *lpOutData***)**

This escape retrieves the number of paper bins available on a printer and sets the current paper bin. See the following "Comments" section for more information on the actions performed by this escape.

Parameter	Type/Description
hDC	**HDC** Identifies the device context.
nCount	**int** Specifies the number of bytes pointed to by the *lpInData* parameter.
lpInData	**BinInfo FAR *** Points to a **BinInfo** data structure that specifies the new paper bin. It may be set to NULL.

Parameter	Type/Description
lpOutData	**BinInfo FAR *** Points to a **BinInfo** data structure that contains information about the current or previous paper bin and the number of bins available.

Comments

There are three possible actions for this escape, depending on the values passed in the *lpInData* and *lpOutData* parameters:

lpInData	*lpOutData*	Action
NULL	**BinInfo**	Retrieves the number of bins and the number of the current bin.
BinInfo	**BinInfo**	Sets the current bin to the number specified in the **BinNumber** field of the data structure to which *lpInData* points and retrieves the number of the previous bin.
BinInfo	NULL	Sets the current bin to the number specified in the **BinNumber** field of the data structure to which *lpInData* points.

The **BinInfo** data structure has the following format:

```
typedef struct{
    DWORD  BinNumber;
    DWORD  NbrofBins;
    DWORD       Reserved;
    DWORD       Reserved;
    DWORD       Reserved;
    DWORD       Reserved;
} BinInfo;
```

The **BinInfo** structure has the following fields:

Field	Description
BinNumber	Identifies the current or previous paper bin.
NbrofBins	Specifies the number of paper bins available.

When setting a new bin, the setting does not take effect until a new device context is created (without initialization data). The setting will take immediate effect if the high bit of the bin number is set, so that the next page printed will come from the new bin. For example, 0x8001 uses the second bin immediately whenever 0x0001 sets the same bin as the default for later print jobs.

In general, only the immediate-selection form should be used by applications. Setting the bin for future print jobs is supported for backward compatibility to an earlier form of this escape which appeared in some versions of HP's Page Control Language (PCL) and PostScript.

GETSETPAPERMETRICS

Syntax

short Escape(*hDC***, GETSETPAPERMETRICS, sizeof(RECT),** *lpNewPaper*, *lpPrevPaper***)**

This escape sets the paper type according to the given paper metrics information. It also retrieves the current printer's paper metrics information.

This escape expects a **RECT** data structure representing the imageable area of the physical page and assumes the origin is in the upper-left corner.

Parameter	Type/Description
hDC	**HDC** Identifies the device context.
lpNewPaper	**LPRECT** Points to a **RECT** data structure that defines the new imageable area.
lpPrevPaper	**LPRECT** Points to a **RECT** data structure that receives the previous imageable area.

Return Value

The return value is positive if successful, zero if the escape is not implemented, and negative if an error occurs.

Comments

This escape is provided only for backward compatibility. New applications should use the GDI **DeviceCapabilities** and **ExtDeviceMode** functions instead.

GETSETPAPERORIENT

Syntax

short Escape(*hDC***, GETSETPAPERORIENT,** *nCount*, *lpInData*, **NULL)**

This escape returns or sets the current paper orientation.

Parameter	Type/Description
hDC	**HDC** Identifies the device context.

Parameter	Type/Description
nCount	Specifies the number of bytes pointed to by the *lpInData* parameter.
lpInData	**ORIENT FAR *** Points to an **ORIENT** data structure that specifies the new paper orientation. See the following "Comments" section for a description of this data structure. It may be set to NULL, in which case the **GETSETPAPERORIENT** escape returns the current paper orientation.

Return Value The return value specifies the current orientation if *lpInData* is NULL; otherwise, it is the previous orientation. The return value is –1 if the escape failed.

Comments This escape is provided only for backward compatibility. New applications should use the GDI **DeviceCapabilities** and **ExtDeviceMode** functions instead.

The **ORIENT** data structure has the following format:

```
typedef struct{
    DWORD  Orientation;
    DWORD  Reserved;
    DWORD  Reserved;
    DWORD  Reserved;
    DWORD  Reserved;
} ORIENT;
```

The **Orientation** field can be either of these values:

Value	Meaning
1	The new orientation is portrait.
2	The new orientation is landscape.

This escape is also known as **GETSETPAPERORIENTATION.**

GETSETSCREENPARAMS

Syntax **short Escape(***hDC***, GETSETSCREENPARAMS, sizeof(SCREENPARAMS),** *lpInData***,** *lpOutData***)**

This escape retrieves or sets the current screen information for rendering halftones.

Parameter	Type/Description
hDC	**HDC** Identifies the device context.
lpInData	**SCREENPARAMS FAR *** Points to a **SCREENPARAMS** data structure that contains the new screen information. This parameter may be NULL.
lpOutData	**SCREENPARAMS FAR *** Points to a **SCREENPARAMS** data structure that retrieves the previous screen information. This parameter may be NULL.

Return Value

The return value specifies the outcome of the escape. It is positive if the escape is successful. Otherwise, it is negative.

Comments

This escape affects how device-independent bitmaps (DIBs) are rendered and how color objects are filled.

The **SCREENPARAMS** data structure has the following format:

```
typedef struct {
  int    angle;
  int    frequency;
  DWORD  types;
} SCREENPARAMS;
```

The **SCREENPARAMS** structure has the following fields:

Field	Description
angle	Specifies in degrees the angle of the halftone screen.
frequency	Specifies in dots per inch of the screen frequency.
types	Is a mask containing bits which indicate the type of screen cell. If a pointer to this structure is passed as the *lpInData* parameter, only one bit may be set. If the *lpOutData* parameter contains a pointer to this structure, when the escape returns, the *types* field will have a bit set for each type supported by the printer driver. Acceptable bit values are:

- DIAMOND
- DOT
- ELLIPSE

- LINE

GETTECHNOLOGY

Syntax **short Escape(*hDC*, GETTECHNOLOGY, NULL, NULL, *lpTechnology*)**

This escape retrieves the general technology type for a printer, thereby allowing an application to perform technology-specific actions.

Parameter	Type/Description
hDC	**HDC** Identifies the device context.
lpTechnology	**LPSTR** Points to a buffer to which the driver copies a null-terminated string containing the printer technology type, such as "PostScript."

Return Value The return value specifies the outcome of the escape. It is 1 if the escape is successful, and is zero if the escape is not successful or is not implemented.

GETTRACKKERNTABLE

Syntax **short Escape(*hDC*, GETTRACKKERNTABLE, NULL, NULL, *lpOutData*)**

This escape fills the buffer pointed to by the *lpOutData* parameter with the track-kerning table for the currently selected font.

Parameter	Type/Description
hDC	**HDC** Identifies the device context.
lpOutdata	**KERNTRACK FAR *** Points to an array of **KERNTRACK** structures. This array must be large enough to accommodate all the font's kerning tracks. The number of tracks in the font can be obtained from the **EXTTEXTMETRIC** structure returned by the **GETEXTENDEDTEXTMETRICS** escape. See the following "Comments" section for the format of the **KERNTRACK** data structure.

Return Value The return value specifies the number of **KERNTRACK** structures copied to the buffer. This value is zero if the font does not have kerning tracks defined, or if the escape fails or is not implemented.

Comments The **KERNTRACK** data structure has the following format:

```
typedef struct {
   short     ktDegree;
   short     ktMinSize;
   short     ktMinAmount;
   short     ktMaxSize;
   short     ktMaxAmount;
   } KERNTRACK;
```

The **KERNTRACK** structure contains the following fields:

Field	Description
ktDegree	Specifies the amount of track kerning. Increasingly negative values represent tighter track kerning, and increasingly positive values represent looser track kerning.
ktMinSize	Specifies in device units the minimum font size for which linear track kerning applies.
ktMinAmount	Specifies in font units the amount of track kerning to apply to font sizes less than or equal to the size specified by the **ktMinSize** field.
ktMaxSize	Specifies in device units the maximum font size for which linear track kerning applies.
ktMaxAmount	Specifies in font units the amount of track kerning to apply to font sizes greater than or equal to the size specified by the **ktMaxSize** field.

Between the **ktMinSize** and **ktMaxSize** font sizes, track kerning is a linear function from **ktMinAmount** to **ktMaxAmount**. The values returned in the **KERNTRACK** structures are affected by whether relative character widths are enabled or disabled. For more information, see the description of the **ENABLERELATIVEWIDTHS** escape earlier in this chapter.

GETVECTORBRUSHSIZE

Syntax

short Escape(*hDC***, GETVECTORBRUSHSIZE, sizeof(LOGBRUSH),** *lpInData*, *lpOutData***)**

This escape retrieves in device units the size of a plotter pen used to fill closed figures. GDI uses this information to prevent the plotter pen from writing over the borders of the figure when filling closed figures.

Parameter	Type/Description
hDC	**HDC** Identifies the device context.
lpInData	**LOGBRUSH FAR *** Points to a **LOGBRUSH** data structure that specifies the brush for which data are to be returned.
lpOutData	**LPPOINT** Points to a **POINT** data structure that contains in its second word the width of the pen in device units.

Return Value

The return value specifies the outcome of the escape. It is 1 if the escape is successful; it is zero if the escape is not successful or is not implemented.

GETVECTORPENSIZE

Syntax

short Escape(*hDC***, GETVECTORPENSIZE, sizeof(LOGPEN),** *lpInData*, *lpOutData***)**

This escape retrieves the size in device units of a plotter pen. GDI uses this information to prevent hatched brush patterns from overwriting the border of a closed figure.

Parameter	Type/Description
hDC	**HDC** Identifies the device context.
lpInData	**LOGPEN FAR *** Points to a **LOGPEN** data structure that specifies the pen for which the width is to be retrieved.
lpOutData	**LPPOINT** Points to a **POINT** data structure that contains in its second word the width of the pen in device units.

Return Value

The return value specifies the outcome of the escape. It is 1 if the escape is successful; it is zero if the escape is not successful or if it is not implemented.

MFCOMMENT

Syntax

BOOL **Escape(***hDC***, MFCOMMENT,** *nCount*, *lpComment*, **NULL)**

This escape adds a comment to a metafile.

Parameter	Type/Description
hDC	**HDC** Identifies the device context for the device on which the metafile appears.
nCount	**short** Specifies the number of characters in the string pointed to by the *lpComment* parameter.
lpComment	**LPSTR** Points to a null-terminated string that contains the comment that will appear in the metafile.

Return Value

The return value specifies the outcome of the escape. It is –1 if an error such as insufficient memory or an invalid port specification occurs. Otherwise, it is positive.

NEWFRAME

Syntax

short **Escape(***hDC***, NEWFRAME, NULL, NULL, NULL)**

This escape informs the device that the application has finished writing to a page. This escape is typically used with a printer to direct the device driver to advance to a new page.

Parameter	Type/Description
hDC	**HDC** Identifies the device context.

Return Value

The return value specifies the outcome of the escape. It is positive if the escape is successful. Otherwise, it is one of the following values:

Value	Meaning
SP_APPABORT	Job was terminated because the application's abort function returned zero.
SP_ERROR	General error.

Value	Meaning
SP_OUTOFDISK	Not enough disk space is currently available for spooling, and no more space will become available.
SP_OUTOFMEMORY	Not enough memory is available for spooling.
SP_USERABORT	User terminated the job through the Print Manager.

Comments

Do not use the **NEXTBAND** escape with **NEWFRAME**. For banding drivers, GDI replays a metafile to the printer, simulating a sequence of **NEXTBAND** escapes.

The **NEWFRAME** escape restores the default values of the device context. Consequently, if a font other than the default font is selected when the application calls the **NEWFRAME** escape, the application must select the font again following the **NEWFRAME** escape.

NEXTBAND

Syntax

short Escape(*hDC*, **NEXTBAND, NULL, NULL,** *lpBandRect*)

This escape informs the device driver that the application has finished writing to a band, causing the device driver to send the band to the Print Manager and return the coordinates of the next band. Applications that process banding themselves use this escape.

Parameter	Type/Description
hDC	**HDC** Identifies the device context.
lpBandRect	**LPRECT** Points to the **RECT** data structure that will receive the next band coordinates. The device driver copies the device coordinates of the next band into this structure.

Return Value

The return value specifies the outcome of the escape. It is positive if the escape is successful. Otherwise, it is one of the following values:

Value	Meaning
SP_APPABORT	Job was terminated because the application's abort function returned zero.
SP_ERROR	General error.

Value	Meaning
SP_OUTOFDISK	Not enough disk space is currently available for spooling, and no more space will become available.
SP_OUTOFMEMORY	Not enough memory is available for spooling.
SP_USERABORT	User terminated the job through the Print Manager.

Comments The **NEXTBAND** escape sets the band rectangle to the empty rectangle when printing reaches the end of a page.

Do not use the **NEWFRAME** escape with **NEXTBAND.**

PASSTHROUGH

Syntax **short Escape(***hDC*, **PASSTHROUGH,** *nCount*, *lpInData*, **NULL)**

This escape allows the application to send data directly to the printer, bypassing the standard print-driver code.

NOTE To use this escape, an application must have thorough knowledge of how the particular printer operates.

Parameter	Type/Description
hDC	**HDC** Identifies the device context.
nCount	**short** Specifies the number of bytes to which the *lpInData* parameter points.
lpInData	**LPSTR** Points to a structure whose first word (16 bits) contains the number of bytes of input data. The remaining bytes of the structure contain the data itself.

Return Value The return value specifies the number of bytes transferred to the printer if the escape is successful. It is less than zero if the escape is not implemented, and less than or equal to zero if the escape is not successful.

Comments There may be restrictions on the kinds of device data an application can send to the device without interfering with the operation of the driver. In general, applications must avoid resetting the printer or causing the page to be printed.

It is strongly recommended that applications not perform functions that consume printer memory, such as downloading a font or a macro.

An application can avoid corrupting its data stream when issuing multiple, consecutive **PASSTHROUGH** escapes if it does not access the printer any other way during the sequence.

QUERYESCSUPPORT

Syntax **short Escape(*hDC*, QUERYESCSUPPORT, sizeof(int), *lpEscNum*, NULL)**

This escape determines whether a particular escape is implemented by the device driver.

Parameter	Type/Description
hDC	**HDC** Identifies the device context.
lpEscNum	**LPINT** Points to a short-integer value that specifies the escape function to be checked.

Return Value The return value specifies whether a particular escape is implemented. It is nonzero for implemented escape functions. Otherwise, it is zero.

If the *lpEscNum* parameter is set to DRAWPATTERNRECT, the return value is one of the following:

Value	Meaning
0	DRAWPATTERNRECT is not implemented.
1	DRAWPATTERNRECT is implemented for a printer other than the HP LaserJet IIP; this printer supports white rules.
2	DRAWPATTERNRECT is implemented for the HP LaserJet IIP.

RESTORE_CTM

Syntax **short Escape(*hDC*, RESTORE_CTM, NULL, NULL, NULL)**

This escape restores the previously saved current transformation matrix.

The current transformation matrix controls the manner in which coordinates are translated, rotated, and scaled by the device. By using matrices, an application can combine these operations in any order to produce the desired mapping for a particular picture.

Parameter	Type/Description
hDC	**HDC** Identifies the device context.

Return Value The return value specifies the number of **SAVE_CTM** escape calls without a corresponding **RESTORE_CTM** call. If the escape is unsuccessful, the return value is –1.

Comments Applications should not make any assumptions about the initial contents of the current transformation matrix.

This escape uses a matrix specification based on the Microsoft OS/2 Presentation Manager graphics programming interface (GPI) model, which is an integer-coordinate system similar to the system which GDI uses.

SAVE_CTM

Syntax **short Escape(*hDC*, SAVE_CTM, NULL, NULL, NULL)**

This escape saves the current transformation matrix.

The current transformation matrix controls the manner in which coordinates are translated, rotated, and scaled by the device. By using matrices, an application can combine these operations in any order to produce the desired mapping for a particular picture.

An application can restore the matrix by using the **RESTORE_CTM** escape.

An application typically saves the current transformation matrix before changing it. This allows the application to restore the previous state upon completion of a particular operation.

Parameter	Type/Description
hDC	**HDC** Identifies the device context.

Return Value The return value specifies the number of **SAVE_CTM** escape calls without a corresponding **RESTORE_CTM** call. The return value is zero if the escape was unsuccessful.

Comments Applications should not make any assumptions about the initial contents of the current transformation matrix.

Applications are expected to restore the contents of the current transformation matrix.

This escape uses a matrix specification based on the OS/2 Presentation Manager graphics programming interface (GPI) model, which is an integer-coordinate system similar to the system that GDI uses.

SELECTPAPERSOURCE

This escape has been superseded by the **GETSETPAPERBINS** escape and is provided only for backward compatibility. New applications should use the **GETSETPAPERBINS** escape instead.

SETABORTPROC

Syntax

short Escape(*hDC*, SETABORTPROC, NULL, *lpAbortFunc*, NULL)

This escape sets the abort function for the print job.

If an application is to allow the print job to be canceled during spooling, it must set the abort function before the print job is started with the **STARTDOC** escape. Print Manager calls the abort function during spooling to allow the application to cancel the print job or to process out-of-disk-space conditions. If no abort function is set, the print job will fail if there is not enough disk space for spooling.

Parameter	Type/Description
hDC	**HDC** Identifies the device context.
lpAbortFunc	**FARPROC** Points to the application-supplied abort function. See the following "Comments" section for details.

Return Value

The return value specifies the outcome of the escape. It is positive if the escape is successful. Otherwise, it is negative.

Comments

The address passed as the *lpAbortFunc* parameter must be created by using the **MakeProcInstance** function.

The callback function must use the Pascal calling convention and must be declared **FAR**. The abort function must have the following form:

Callback Function

short FAR PASCAL *AbortFunc*(*hPr*, *code*)
HDC hPr;
short code;

AbortFunc is a placeholder for the application-supplied function name. The actual name must be exported by including it in an **EXPORTS** statement in the application's module-definition file.

Parameter	Description
hPr	Identifies the device context.
code	Specifies whether an error has occurred. It is zero if no error has occurred. It is SP_OUTOFDISK if Print Manager is currently out of disk space and more disk space will become available if the application waits. If *code* is SP_OUTOFDISK, the application does not have to abort the print job. If it does not, it must yield to Print Manager by calling the **PeekMessage** or **GetMessage** function.

Return Value

The return value should be nonzero if the print job is to continue, and zero if it is canceled.

SETALLJUSTVALUES

Syntax **short Escape(***hDC***, SETALLJUSTVALUES, sizeof(JUST_VALUE_STRUCT),** *lpInData***, NULL)**

This escape sets all of the text-justification values that are used for text output.

Text justification is the process of inserting extra pixels among break characters in a line of text. The blank character is normally used as a break character.

Parameter	Type/Description
hDC	**HDC** Identifies the device context.
lpInData	**JUST_VALUE_STRUCT FAR *** Points to a **JUST_VALUE_STRUCT** data structure that defines the text-justification values. See the following "Comments" section for more information on the **JUST_VALUE_STRUCT** data structure.

Return Value The return value specifies the outcome of the escape. It is 1 if the escape is successful. Otherwise, it is zero.

Comments The *lpInData* parameter points to a **JUST_VALUE_STRUCT** data structure that describes the text-justification values used for text output. The **JUST_VALUE_STRUCT** structure has the following format:

```
typedef struct {
    short   nCharExtra;
    WORD    nCharCount;
    short   nBreakExtra;
    WORD    nBreakCount;
} JUST_VALUE_STRUCT;
```

This structure has the following fields:

Field	Description
nCharExtra	Specifies the total extra space (in font units) that must be distributed over **nCharCount** characters.
nCharCount	Specifies the number of characters over which **nCharExtra** is distributed.
nBreakExtra	Specifies the total extra space (in font units) that is distributed over **nBreakCount** characters.
nBreakCount	Specifies the number of break characters over which **nBreakExtra** is distributed.

The units used for **nCharExtra** and **nBreakExtra** are the font units of the device and are dependent on whether relative character widths were enabled with the **ENABLERELATIVEWIDTHS** escape.

The values set with this escape apply to subsequent calls to the **TextOut** function. The driver stops distributing the extra space specified in the **nCharExtra** field when it has output the number of characters specified in the **nCharCount** field. Likewise, it stops distributing the space specified by the **nBreakExtra** field when it has output the number of characters specified by the **nBreakCount** field. A call on the same string to the **GetTextExtent** function made immediately after the call to the **TextOut** function will be processed in the same manner.

To re-enable justification with the **SetTextJustification** and **SetTextCharacterExtra** functions, an application should call the **SETALLJUSTVALUES** escape and set the **nCharExtra** and **nBreakExtra** fields to zero.

SET_ARC_DIRECTION

Syntax short **Escape(***hDC*, **SET_ARC_DIRECTION, sizeof(int),** *lpDirection*, **NULL)**

This escape specifies the direction in which elliptical arcs are drawn using the GDI **Arc** function.

By convention, elliptical arcs are drawn counterclockwise by GDI. This escape lets an application draw paths containing arcs drawn clockwise.

Parameter	Type/Description
hDC	**HDC** Identifies the device context.
lpDirection	**LPINT** Points to a short integer specifying the arc direction. It can be either of the following values: ■ COUNTERCLOCKWISE (0) ■ CLOCKWISE (1)

Return Value The return value is the previous arc direction.

Comments This escape maps to PostScript language elements and is intended for PostScript line devices.

SET_BACKGROUND_COLOR

Syntax **short Escape(***hDC*, **SET_BACKGROUND_COLOR,** *nCount*, *lpNewColor*, *lpOldColor*)

This escape sets and retrieves the current background color for the device.

The background color is the color of the display surface before an application draws anything on the device. This escape is particularly useful for color printers and film recorders.

This escape should be sent before the application draws anything on the current page.

Parameter	Type/Description
hDC	**HDC** Identifies the device context.
nCount	**int** Specifies the number of bytes pointed to by the *lpNewColor* parameter.
lpNewColor	**DWORD FAR *** Points to a 32-bit integer specifying the desired background color. This parameter can be NULL if the application is merely retrieving the current background color.
lpOldColor	**DWORD FAR *** Points to a 32-bit integer which receives the previous background color. This parameter can be NULL if the application does not use the previous background color.

Return Value The return value is TRUE if the escape was successful and FALSE if it was unsuccessful.

Comments The default background color is white.

The background color is reset to the default when the device driver receives an **ENDDOC** or **ABORTDOC** escape.

SET_BOUNDS

Syntax **short Escape(*hDC*, SET_BOUNDS, sizeof(RECT), *lpInData*, NULL)**

This escape sets the bounding rectangle for the picture being produced by the device driver supporting the given device context. It is used when creating images in a file format such as Encapsulated PostScript (EPS) and Hewlett-Packard Graphics Language (HPGL) for which there is a device driver.

Parameter	Type/Description
hDC	**HDC** Identifies the device context.
lpInData	**LPRECT** Points to a **RECT** data structure that specifies in device coordinates a rectangle that bounds the image to be output.

Return Value The return value is TRUE if the escape was successful; otherwise, the return value is FALSE.

Comments An application should issue this escape before each page in the image. For single-page images, this escape should be issued immediately before the **STARTDOC** escape.

When an application uses coordinate-transformation escapes, device drivers may not perform bounding box calculations correctly. When an application uses the **SET_BOUNDS** escape, the driver does not have to calculate the bounding box.

Applications should always use this escape to ensure support for the Encapsulated PostScript (EPS) printing capabilities that will be built into future PostScript drivers.

SETCOLORTABLE

Syntax

short Escape(*hDC*, SETCOLORTABLE, sizeof(COLORTABLE_STRUCT), *lpInData*, *lpColor*)

This escape sets an RGB color-table entry. If the device cannot supply the exact color, the function sets the entry to the closest possible approximation of the color.

Parameter	Type/Description
hDC	**HDC** Identifies the device context.
lpInData	**COLORTABLE_STRUCT FAR *** Points to a **COLORTABLE_STRUCT** data structure that contains the index and RGB value of the color-table entry. See the following "Comments" section for more information on the **COLORTABLE_STRUCT** data structure.
lpColor	**DWORD FAR *** Points to the long-integer value that is to receive the RGB color value selected by the device driver to represent the requested color value.

Return Value

The return value specifies the outcome of the escape. It is positive if the escape is successful. Otherwise, it is negative.

Comments

The **COLORTABLE_STRUCT** data structure has the following format:

```
typedef struct {
    WORD   Index;
    DWORD  rgb;
} COLORTABLE_STRUCT;
```

This structure has the following fields:

Field	Description
Index	Specifies the color-table index. Color-table entries start at zero for the first entry.
rgb	Specifies the desired RGB color value.

A device's color table is a shared resource; changing the system display color for one window changes it for all windows. Only applications developers who have a thorough knowledge of the display driver should use this escape.

The **SETCOLORTABLE** escape has no effect on devices with fixed color tables.

This escape is intended for use by both printer and display drivers. However, the EGA and VGA color drivers do not support it.

This escape changes the palette used by the display driver. However, since the driver's color-mapping algorithms will probably no longer work with a different palette, an extension has been added to this function.

If the color index pointed to by the *lpInData* parameter is 0XFFFF, the driver is to leave all color-mapping functionality to the calling application. The application must use the proper color-mapping algorithm and take responsibility for passing the correctly mapped physical color to the driver (instead of the logical RGB color) in such device-driver functions as **RealizeObject** and **ColorInfo**.

For example, if the device supports 256 colors with palette indexes of 0 through 255, the application would determine which index contains the color that it wants to use in a certain brush. It would then pass this index in the low-order byte of the **DWORD** logical color passed to the **RealizeObject** device-driver function. The driver would then use this color exactly as passed instead of performing its usual color-mapping algorithm. If the application wants to reactivate the driver's color-mapping algorithm (that is, if it restores the original palette when switching from its window context), then the color index pointed to by *lpInData* should be 0xFFFE.

SETCOPYCOUNT

Syntax

short Escape(*hDC*, **SETCOPYCOUNT, sizeof(int),** *lpNumCopies*, *lpActualCopies*)

This escape specifies the number of uncollated copies of each page that the printer is to print.

Parameter	Type/Description
hDC	**HDC** Identifies the device context.
lpNumCopies	**LPINT** Points to a short-integer value that contains the number of uncollated copies to be printed.
lpActualCopies	**LPINT** Points to a short-integer value that will receive the number of copies to be printed. This may be less than the number requested if the requested number is greater than the device's maximum copy count.

Return Value

The return value specifies the outcome of the escape. It is 1 if the escape is successful; it is zero if the escape is not successful. If the escape is not implemented, the return value is zero.

SETKERNTRACK

Syntax

short Escape(*hDC***, SETKERNTRACK, sizeof(int),** *lpNewTrack*, *lpOldTrack*)

This escape specifies which kerning track to use for drivers that support automatic track kerning. A kerning track of zero disables automatic track kerning.

When track kerning is enabled, the driver will automatically kern all characters according to the specified track. The driver will reflect this kerning both on the printer and in **GetTextExtent** function calls.

Parameter	Type/Description
hDC	**HDC** Identifies the device context.
lpNewTrack	**LPINT** Points to a short-integer value that specifies the kerning track to use. A value of zero disables this feature. Values in the range 1 to *nKernTracks* correspond to positions in the track-kerning table (using 1 as the first item in the table). For more information, see the description of the **EXTTEXTMETRIC** structure provided under the description of the **GETEXTENDEDTEXTMETRICS** escape.
lpOldTrack	**LPINT** Points to a short-integer value that will receive the previous kerning track.

Return Value

The return value specifies the outcome of the escape. It is 1 if the escape is successful; it is zero if the escape is not successful or not implemented.

Comments

Automatic track kerning is disabled by default.

A driver does not have to support the **ENABLEPAIRKERNING** escape just because it supplies the track-kerning table to the application by using the **GETTRACKKERNTABLE** escape. In the case where **GETTRACKKERNTABLE** is supported but the **SETKERNTRACK** escape is not, the application must properly space the characters on the output device.

SETLINECAP

Syntax **short Escape(***hDC***, SETLINECAP, sizeof(int),** *lpNewCap*, *lpOldCap*)

This escape sets the line end cap.

A line end cap is that portion of a line segment that appears on either end of the segment. The cap may be square or circular. It can extend past, or remain flush with the specified segment end points.

Parameter	Type/Description
hDC	**HDC** Identifies the device context.
lpNewCap	**LPINT** Points to a short-integer value that specifies the end-cap type. The possible values and their meanings are given in the following list:

Value	Meaning
–1	Line segments are drawn by using the default GDI end cap.
0	Line segments are drawn with a squared end point that does not project past the specified segment length.
1	Line segments are drawn with a rounded end point; the diameter of this semicircular arc is equal to the line width.
2	Line segments are drawn with a squared end point that projects past the specified segment length. The projection is equal to half the line width.

lpOldCap	**LPINT** Points to a short-integer value that specifies the previous end-cap setting.

Return Value The return value specifies the outcome of the escape. It is positive if the escape is successful. Otherwise, it is negative.

Comments The interpretation of this escape varies with page-description languages (PDLs). Consult the PDL documentation for its exact meaning.

This escape is also known as **SETENDCAP**.

SETLINEJOIN

Syntax

short Escape(*hDC***, SETLINEJOIN, sizeof(int),** *lpNewJoin***,** *lpOldJoin***)**

This escape specifies how a device driver will join two intersecting line segments. The intersection can form a rounded, squared, or mitered corner.

Parameter	Type/Description
hDC	**HDC** Identifies the device context.
lpNewJoin	**LPINT** Points to a short-integer value that specifies the type of intersection. The possible values and their meanings are given in the following list:

Value	Meaning
–1	Line segments are joined by using the default GDI setting.
0	Line segments are joined with a mitered corner; the outer edges of the lines extend until they meet at an angle. This is referred to as a miter join.
1	Line segments are joined with a rounded corner; a semicircular arc with a diameter equal to the line width is drawn around the point where the lines meet. This is referred to as a round join.
2	Line segments are joined with a squared end point; the outer edges of the lines are not extended. This is referred to as a bevel join.

Parameter	Type/Description
lpOldJoin	**LPINT** Points to a short-integer value that specifies the previous line join setting.

Return Value

The return value specifies the outcome of the escape. It is positive if the escape is successful. Otherwise, it is negative.

Comments

The interpretation of this escape varies with page-description languages (PDLs). Consult the PDL documentation for its exact meaning.

If an application specifies a miter join but the angle of intersection is too small, the device driver ignores the miter setting and uses a bevel join instead.

SETMITERLIMIT

Syntax

short Escape(*hDC*, SETMITERLIMIT, sizeof(int), ***lpNewMiter*, *lpOldMiter*)**

This escape sets the miter limit for a device. The miter limit controls the angle at which a device driver replaces a miter join with a bevel join.

Parameter	Type/Description
hDC	**HDC** Identifies the device context.
nCount	**short** Specifies the number of bytes to which the *lpNewMiter* parameter points.
lpNewMiter	**LPINT** Points to a short-integer value that specifies the desired miter limit. Only values greater than or equal to –1 are valid. If this value is –1, the driver will use the default GDI miter limit.
lpOldMiter	**LPINT** Points to a short-integer value that specifies the previous miter-limit setting.

Return Value

The return value specifies the outcome of the escape. It is positive if the escape is successful. Otherwise, it is negative.

Comments

The miter limit is defined as follows:

$$\frac{\textit{miter length}}{\textit{line width}} = \frac{1}{\sin(x/2)}$$

X is the angle of the line join in radians.

The interpretation of this escape varies with page-description languages (PDLs). Consult the PDL documentation for its exact meaning.

SET_POLY_MODE

Syntax

short Escape(*hDC*, SET_POLY_MODE, sizeof(int), ***lpMode*, NULL)**

This escape sets the poly mode for the device driver. The poly mode is a state variable indicating how to interpret calls to the GDI **Polygon** and **Polyline** functions.

The **SET_POLY_MODE** escape enables a device driver to draw shapes (such as Bezier curves) not supported directly by GDI. This permits applications that draw complex curves to send the curve description directly to a device without having to simulate the curve as a polygon with a large number of points.

Parameter	Type/Description
hDC	**HDC** Identifies the device context.
lpMode	**LPINT** Points to a short integer specifying the desired poly mode. The poly mode is a state variable indicating how points in **Polygon** or **Polyline** function calls should be interpreted. All device drivers are not required to support all possible modes. A device driver returns zero if it does not support the specified mode. The *lpMode* parameter may be one of the following values:

Value	Meaning
PM_POLYLINE (1)	The points define a conventional polygon or polyline.
PM_BEZIER (2)	The points define a sequence of 4-point Bezier spline curves. The first curve passes through the first four points, with the first and fourth points as end points, and the second and third points as control points. Each subsequent curve in the sequence has the end point of the previous curve as its start point, the next two points as control points, and the third as its end point. The last curve in the sequence is permitted to have fewer than four points. If the curve has only one point, it is considered a point. If it has two points, it is a line segment. If it has three points, it is a parabola defined by drawing a Bezier curve with the first and third points as end points and the two control points equal to the second point.
PM_POLYLINE-SEGMENT (3)	The points specify a list of coordinate pairs. Line segments are drawn connecting each successive pair of points.

Return Value

The return value is the previous poly mode. If the return value is zero, the device driver did not handle the request.

Comments

An application should issue the **SET_POLY_MODE** escape before it draws a complex curve. It should then call the **Polyline** or **Polygon** function with the desired control points defining the curve. After drawing the curve, the application should reset the driver to its previous state by issuing the **SET_POLY_MODE** escape.

Polyline calls draw using the currently selected pen.

Polygon calls draw using the currently selected pen and brush. If the start and end points are not equal, a line is drawn from the start point to the end point before filling the polygon (or curve).

GDI treats **Polygon** calls using PM_POLYLINESEGMENT mode exactly the same as **Polyline** calls.

Four points define a Bezier curve. GDI generates the curve by connecting the first and second, second and third, and third and fourth points. GDI then connects the midpoints of these consecutive line segments. Finally, GDI connects the midpoints of the lines connecting the midpoints, and so forth.

The line segments drawn in this way converge to a curve defined by the following parametric equations, expressed as a function of the independent variable *t*.

$$X(t) = (1-t)^3 x_1 + 3(1-t)^2 t x_2 + 3(1-t)t^2 x_3 + t^3 x_4$$

$$Y(t) = (1-t)^3 y_1 + 3(1-t)^2 t y_2 + 3(1-t)t^2 y_3 + t^3 y_4$$

The points (x_1,y_1), (x_2,y_2), (x_3,y_3) and (x_4,y_4) are the control points defining the curve. The independent variable *t* varies from 0 to 1.

Primitive types other than PM_BEZIER and PM_POLYLINESEGMENT may be added to this escape in the future. Applications should check the return value from this escape to determine whether or not the driver supports the specified poly mode.

SET_SCREEN_ANGLE

Syntax

short Escape(*hDC*, SET_SCREEN_ANGLE, sizeof(int), *lpAngle*, NULL)

This escape sets the current screen angle to the desired angle and enables an application to simulate the tilting of a photographic mask in producing a color separation for a particular primary.

Parameter	Type/Description
hDC	**HDC** Identifies the device context.
lpAngle	**LPINT** Points to a short-integer value specifying the desired screen angle in tenths of a degree. The angle is measured counterclockwise.

Return Value The return value is the previous screen angle.

Comments Four-color process separation is the process of separating the colors comprising an image into four component primaries: cyan, magenta, yellow, and black. The image is then reproduced by overprinting each primary.

In traditional four-color process printing, half-tone images for each of the four primaries are photographed against a mask tilted to a particular angle. Tilting the mask in this manner minimizes unwanted moire patterns caused by overprinting two or more colors.

The device driver defines the default screen angle.

SET_SPREAD

Syntax **short Escape(*hDC*, SET_SPREAD, sizeof(int), *lpSpread*, NULL)**

This function sets the amount that nonwhite primitives are expanded for a given device to provide a slight overlap between primitives to compensate for imperfections in the reproduction process.

Spot color separation is the process of separating an image into each distinct color used in the image. The image is reproduced by overprinting each successive color in the image.

When reproducing a spot-separated image, the printing equipment must be calibrated to align each page exactly on each pass. However, differences in temperature, humidity, and so forth, between passes often cause images to align imperfectly on subsequent passes. For this reason, lines in spot separations are often widened (spread) slightly to make up for problems in registering subsequent passes through the printer. This process is called trapping. The **SET_SPREAD** escape implements this process.

Parameter	Type/Description
hDC	**HDC** Identifies the device context.
lpSpread	**LPINT** Points to a short-integer value that specifies the amount, in pixels, by which all nonwhite primitives are to be expanded.

Return Value The return value is the previous spread value.

Comments The default spread is zero.

The current spread applies to all bordered primitives (whether or not the border is visible) and text.

STARTDOC

Syntax **short Escape(***hDC***, STARTDOC,** *nCount*, *lpDocName*, **NULL)**

This escape informs the device driver that a new print job is starting and that all subsequent **NEWFRAME** escape calls should be spooled under the same job until an **ENDDOC** escape call occurs. This ensures that documents longer than one page will not be interspersed with other jobs.

Parameter	Type/Description
hDC	**HDC** Identifies the device context.
nCount	**short** Specifies the number of characters in the string pointed to by the *lpDocName* parameter.
lpDocName	**LPSTR** Points to a null-terminated string that specifies the name of the document. The document name is displayed in the Print Manager window. The maximum length of this string is 31 characters plus the terminating null character.

Return Value The return value specifies the outcome of the escape. It is –1 if an error such as insufficient memory or an invalid port specification occurs. Otherwise, it is positive.

Comments The correct sequence of events in a printing operation is as follows:

1. Create the device context.
2. Set the abort function to keep out-of-disk-space errors from terminating a printing operation.

   An abort procedure that handles these errors must be set by using the **SETABORT-PROC** escape.
3. Begin the printing operation with the **STARTDOC** escape.
4. Begin each new page with the **NEWFRAME** escape, or each new band with the **NEXTBAND** escape.

5. End the printing operation with the **ENDDOC** escape.
6. Destroy the cancel dialog box, if any.
7. Free the procedure-instance address of the abort function.

If an application encounters a printing error or a canceled print operation, it must not attempt to terminate the operation by using the **Escape** function with either the **ENDDOC** or **ABORTDOC** escape. GDI automatically terminates the operation before returning the error value.

TRANSFORM_CTM

Syntax

short Escape(*hDC*, TRANSFORM_CTM, 36, *lpMatrix*, NULL)

This escape modifies the current transformation matrix. The current transformation matrix controls the manner in which coordinates are translated, rotated, and scaled by the device. By using matrices, you can combine these operations in any order to produce the desired mapping for a particular picture.

The new current transformation matrix will contain the product of the matrix referenced by the *lpMatrix* parameter and the previous current transformation matrix (CTM = M × CTM).

Parameter	Type/Description
hDC	**HDC** Identifies the device context.
lpMatrix	**LPSTR** Points to a 3-by-3 array of 32-bit integer values specifying the new transformation matrix. Entries in the matrix are scaled to represent fixed-point real numbers. Each matrix entry is scaled by 65,536. The high-order word of the entry contains the whole integer portion, and the low-order word contains the fractional portion.

Return Value

The return value is TRUE if the escape was successful and FALSE if it was unsuccessful.

Comments

Applications should not make any assumptions about the initial value of the current transformation matrix.

The matrix specification used for this escape is based on the Microsoft OS/2 Presentation Manager graphics programming interface (GPI) model, which is an integer-coordinate system similar to the one used by GDI.

Chapter 13 Assembly-Language Macros Overview

Assembly-language Microsoft Windows applications are highly structured assembly-language programs that use high-level-language calling conventions as well as Windows functions, data types, and programming conventions. Although you create assembly-language Windows programs by using the Microsoft Macro Assembler, the goal is to generate object files that are similar to the object files generated by the C Compiler. This chapter gives some guidelines that can help you meet this goal when creating assembly-language Windows applications.

The SDK includes the file CMACROS.INC. This file contains high-level-language macros that define segments, programming models, function interfaces, and data types needed to create Windows applications. The Cmacros provide assembly-time options that define the memory model and the calling conventions that the application will use. The options must be selected in the assembly-language source file prior to the **INCLUDE** directive.

This chapter provides an overview of the Cmacros and supplies important information on creating an assembly-language Windows application. The chapter covers the following topics:

- How to create an assembly-language Windows application
- An overview of the Cmacros
- How to use the Cmacros in an assembly-language application

13.1 Guidelines for Creating Assembly-Language Windows Applications

When creating an assembly-language Windows application using the Cmacros, you should add the following to your application's assembly-language source file:

1. Specify the memory model by setting one of the options **memS**, **memM**, **memC**, or **memL** to 1.
2. Specify the Pascal calling convention by setting the **?PLM** option to 1.

   This is required for functions that will be called by Windows.

3. Enable the Windows prolog and epilog by setting the **?WIN** option to 1.

   This is required for callback functions or for exported functions in Windows libraries.

4. Include the CMACROS.INC file in the application source file.

   The statement that includes the CMACROS.INC file must come after the statements described in the preceding steps.

5. Create the application entry point, WinMain, and make sure that it is declared a public function.

6. Declare any callback functions as described in Section 13.1.6, "Declaring Callback Functions."

7. After assembling the application source files, link your application's assembled object files with the appropriate C-language library for Windows and C run-time libraries.

The rest of this section describes these steps in more detail.

13.1.1 Specifying a Memory Model

The Cmacro memory-model options specify the memory model that the application will use. The memory model defines how many code and data segments are in the application. The following is a list of the possible memory models:

Model	Description
Small	One code segment and one data segment.
Medium	Multiple code segments and one data segment.
Compact	One code segment and multiple data segments.
Large	Multiple code and data segments.
Huge	Multiple code segments and multiple data segments with one or more data items larger than 64K.

Select a memory model by defining the option name at the beginning of the assembly-language source file. Table 13.1 shows the option names available:

Table 13.1 Memory Options

Option Name	Memory Model	Code Size	Data Size
memS	small	small	small
memM	medium	large	small
memC	compact	small	large
memL	large	large	large
memH	huge	large	large

You can define a name by using the **EQU** directive. The definition has the following form:

```
memM      EQU      1
```

If no option is selected, the default model is small.

When you select a memory-model option, two symbols are defined. These two symbols can be used for code that is dependent on the memory model:

SizeC	0 = small code 1 = large code
SizeD	0 = small data 1 = large data 2 = huge data

13.1.2 Selecting a Calling Convention

The Cmacro calling-convention option specifies the high-level-language calling convention that the application will use. You can select the calling convention by defining the value of the symbol **?PLM**. Table 13.2 lists the values and conventions.

Table 13.2 Calling Conventions

?PLM value	Convention	Description
0	Standard C	The caller pushes the rightmost argument onto the stack first, the leftmost last. The caller pops the arguments off the stack after control is returned.
1	Pascal	The caller pushes the leftmost argument onto the stack first, the rightmost last. The called function pops the arguments off the stack.

You can set the **?PLM** symbol value by using the = directive. The statement has the following form:

```
?PLM = 1
```

The default is the Pascal calling convention. The Pascal calling convention is required for functions that are called by Windows.

13.1.3 Enabling the Windows Prolog/Epilog Option

The Windows prolog/epilog option is required for Windows applications. It specifies whether to use special prolog and epilog code with each function; this code defines the current data segment for the given function.

You set this option by defining the value of the symbol **?WIN**. Table 13.3 lists the values:

Table 13.3 Prolog/Epilog Code Options

?WIN value	Meaning
0	Disables the special prolog/epilog code.
1	Enables the special prolog/epilog code.

You can set the **?WIN** symbol value by using the = directive. The statement has the following form:

```
?WIN = 1
```

By default, the prolog and epilog code are enabled.

13.1.4 Including the File CMACROS.INC

The file CMACROS.INC contains the assembly-language definitions for all the Cmacro macros. You must include this file at the beginning of the assembly-language source file by using the **INCLUDE** directive. The line has the following form:

```
INCLUDE CMACROS.INC
```

You must give the full pathname if the macro file is not in the current directory or in a directory specified on the command line.

For a complete description of each of the Cmacro macros, see Chapter 14, "Assembly-Language Macros Directory."

13.1.5 Creating the Application Entry Point

Create the application entry point, WinMain, and make sure that it is declared a public function. It should have the following form:

```
cProc WinMain, <PUBLIC>, <si,di>
     parmW hInstance
     parmW hPrevInstance
     parmD lpCmdLine
     parmW nCmdShow
cBegin WinMain
     .
     .
     .
cEnd WinMain

sEnd
```

The WinMain function should be defined within the standard code segment **CODE**.

13.1.6 Declaring Callback Functions

Make sure any callback functions are declared as follows:

```
cProc TestWndProc, <FAR,PUBLIC>, <si,di>
     parmW hWnd
     parmW message
     parmW wParam
     parmD lParam
cBegin TestWndProc
     .
     .
     .
cEnd TestWndProc
```

Callback functions must be defined within a code segment.

13.1.7 Linking with Libraries

After assembling your application's source files, you should link the assembled object files with the appropriate C-language libraries.

If the entire application is written in assembly language, to link properly you may need to add an external definition for the absolute symbol **__acrtused** in your application source file.

13.1.8 Enabling Stack Checking

You can enable stack checking by defining the symbol **?CHKSTK**. When stack checking is enabled, the prolog code calls the externally defined routine **CHKSTK** to allocate local variables.

You can define the **?CHKSTK** symbol by using the = directive. The statement has the following form:

```
?CHKSTK = 1
```

Once **CHKSTK** is defined, stack checking is enabled for the entire file.

The default (when **CHKSTK** is not defined) is no stack checking.

13.2 Cmacro Groups

Chapter 14, "Assembly-Language Macros Directory," lists and describes the Cmacro macros, a set of assembly-language macros that can be used with the Microsoft Macro Assembler (**MASM**) to create assembly-language Windows applications. The Cmacros provide a simplified interface to the function and segment conventions of high-level languages such as C.

The Cmacros are divided into the following groups:

- Segment macros
- Storage-allocation macros
- Function macros
- Call macros
- Special-definition macros
- Error macros

The rest of this section briefly describes each group of macros.

13.2.1 Segment Macros

Segment macros give access to the code and data segments that an application will use. These segments have the names, attributes, classes, and groups required by Windows.

The Cmacros have two predefined segments, named **CODE** and **DATA**, that any application can use without special definition.

Macro Name	Description
createSeg	Creates a new segment that has the specified name and segment attributes.
sBegin	Opens up a segment (this macro is similar to the **SEGMENT** assembler directive).
sEnd	Closes a segment (this macro is similar to the **ENDS** assembler directive).
assumes	Makes all references to data and code in the segment *segName* relative to the segment register given by *segReg*. It is similar to the **ASSUME** assembler directive.
dataOFFSET	Generates an offset relative to the start of the group to which the **DATA** segment belongs. It is similar to the **OFFSET** assembler operator, but automatically provides the group name.
codeOFFSET	Generates an offset relative to the start of the group to which the **CODE** segment belongs. It is similar to the **OFFSET** assembler operator, but automatically provides the group name.
*segName***OFFSET**	Generates an offset relative to the start of the group to which the user-defined segment *segName* belongs. It is similar to the **OFFSET** assembler operator, but automatically provides the group name.

13.2.2 Storage-Allocation Macros

Storage-allocation macros allocate static memory (either private or public), declare externally defined memory and procedures, and allow the definition of public labels.

Macro Name	Description
static*X*	Allocates private static-memory storage.
global*X*	Allocates public static-memory storage.
extern*X*	Defines one or more names that will be the labels of external variables or functions.
label*X*	Defines one or more names that will be the labels of public (global) variables or functions.

13.2.3 Function Macros

Function macros define the names, attributes, parameters, and local variables of functions.

Macro Name	Description
cProc	Defines the name and attributes of a function.
parm*X*	Defines one or more function parameters. The parameters provide access to the arguments passed to the function.
local*X*	Defines one or more frame variables for the specified function.
cBegin	Defines the actual entry point for the specified function.
cEnd	Defines the exit point for the specified function.

13.2.4 Call Macros

Call macros can be used to call **cProc** functions and high-level-language functions. These macros pass arguments according to the calling convention defined by the **?PLM** option.

Macro Name	Description
cCall	Pushes the specified arguments onto the stack, saves registers (if any), and calls the specified function.
Save	Directs the next **cCall** macro to save the specified registers on the stack before calling a function, and to restore the registers after the function returns.
Arg	This macro defines the arguments to be passed to a function by the next **cCall** macro.

13.2.5 Special-Definition Macros

Special-definition macros inform the Cmacros about user-defined variables, function-register use, and register pointers.

Macro Name	Description
Def	Registers the name of a user-defined variable with the Cmacros.
FarPtr	Defines a 32-bit pointer value that can be passed as a single argument in a **cCall** macro.

13.2.6 Error Macros

Error macros generate an error message to the console and an error message in the listing. Both the text that caused the error and the result of its evaluation are displayed in the generated error message.

Error macros let you code assertions into an assembly-language source program. This lets you code optimum instruction sequences for some operations based on the variable allocation or bit position of flag in a word, and assert that the assumptions made are true.

Macro Name	Description
errnz	Evaluates a given expression. If the result is not zero, an error is displayed.
errn$	Subtracts the offset of the *label* parameter from the offset of the location counter, then adds the *bias* parameter to the result. If this result is not zero, then an error message is displayed.

13.3 Using the Cmacros

This section explains the assembly-language statements generated by some of the Cmacros and illustrates their use with an example of a Cmacros function called **BITBLT**.

13.3.1 Overriding Types

Parameters and local variables created using the **parm***X* and **local***X* macros actually correspond to expressions of the following form:

```
localB x        ==>        x equ byte ptr [bp+nn]
parmB y         ==>        y equ byte ptr [bp+nn]
```

In this example, the *nn* parameter specifies an offset from the current **bp** register value.

These expressions let you use the names without having to explicitly type in "type ptr" and "[bp+offset]" operators. This means that "x" can be referred to as follows:

```
mov     al,x
```

and that "y" can be referred to as follows:

```
mov     ax,y
```

A problem arises if the type must be overridden. The assembler creates an error message if it encounters the following line:

```
mov     ax,word ptr x
```

This can be solved by enclosing the name in parentheses:

```
mov     ax,word ptr (x)
```

One exception to this pattern is the **localV** macro. The expression generated by this macro does not have a type associated with it. Therefore it can be overridden without the parentheses. For example:

```
localV  horse,10   ==>      horse equ [bp+nn]
```

13.3.2 Symbol Redefinition

Any symbol defined by a **parm***X* macro in one function can be redefined as a parameter in any other function. This allows different functions to refer to the same parameter by the same name, regardless of its location on the stack.

13.3.3 Cmacros: a Sample Function

The following example defines the assembly function **BITBLT**, which is a **FAR** and **PUBLIC** type function. When **BITBLT** is invoked, the SI and DI registers are automatically saved, and automatically restored upon exit. The BP register is always saved.

BITBLT is passed seven double-word pointers on the stack. Space will be allocated on the stack for eight frame variables (one structure, five bytes, and two words).

The **cBegin** macro defines the start of the actual code. The *pExt* parameter is loaded, and some values are loaded into registers. The AX and BX registers are saved on the following **cCall**.

Another C function, There, is invoked by the **cCall** macro. Four arguments are passed to There: *pDestBitmap*, the 32-bit pointer in DS:SI, register AX, and register BX. The **cCall** macro places the arguments on the stack in the correct order.

When There returns, the arguments placed on the stack are automatically removed, and the AX and BX registers are restored.

When **cEnd** is reached, the frame variables are removed, any autosave registers are restored, and a return of the correct type (near or far) is performed.

The following example shows how the **BITBLT** function is defined:

```
cProc BITBLT,<FAR,PUBLIC>,<si,di>

parmD   pDestBitmap      ;-> to dest bitmap descriptor
parmD   pDestOrg         ;-> to dest origin (a point)
parmD   pSrcBitmap       ;-> to source bitmap descriptor
parmD   pSrcOrg          ;-> to source origin
parmD   pExt             ;-> to rectangle extent
parmD   pRop             ;-> to rasterop descriptor
parmD   pBrush           ;-> to a physical brush

localV  nOps,4           ;# of each operand used

localB  phaseH           ;Horizontal phase (rotate count)
localB  PatRow           ;Current row for patterns [Ø..7]
localB  direction        ;Increment/decrement flag

localW  startMask        ;mask for first dest byte
localW  lastMask         ;mask for last dest byte

localB  firstFetch       ;Number of first fetches needed
localB  stepDirection    ;Direction of move (left, right)

cBegin

lds     si,pExt
mov     ax,extentX[si]
mov     bx,extentY[si]

RegPtr  dest,ds,si
Save    <ax,bx>

cCall   THERE,<pDestBitmap,dest,ax,bx>

mov     extentX[si],cx
mov     extentY[si],dx

\.
\.
\.
cEnd
```

13.4 Summary

The CMACROS.INC file defines segments, programming models, function interfaces, and data types needed to create Windows applications. The Cmacros provide assembly-time options that define the memory model and the calling conventions that the application will use. For more information on topics related to the Cmacros, see the following:

Topic	Reference
Cmacro descriptions	*Reference, Volume 2*: Chapter 14, "Assembly-Language Macros Directory"
Using the linker	*Tools*: Chapter 2, "Linking Applications: The Linker"
Using the Macro Assembler	*Microsoft Macro Assembler Programmer's Guide*

Chapter 14

Assembly-Language Macros Directory

This chapter describes the Cmacros, a set of assembly-language macros that can be used with the Microsoft Macro Assembler (**MASM**) to create assembly-language Windows applications. The Cmacros provide a simplified interface to the function and segment conventions of high-level languages such as C.

This section lists the Cmacros in alphabetical order, and describes each macro in detail.

Arg

Syntax

Arg *namelist*

This macro defines the arguments to be passed to a function by the next **cCall** macro. The arguments are pushed onto the stack in the order given. This order must correspond to the order of the function parameters.

More than one **Arg** macro can be given before each **cCall**. Multiple **Arg** macros have the same effect as a single macro.

The *namelist* parameter specifies a list of argument names to be passed to the function. All names must have been previously defined.

Comments

Byte-type parameters are passed as words. There is no sign extension or zeroing of the high-order byte.

Immediate arguments are not supported.

Examples

```
Arg     var1
Arg     var2
Arg     var3
Arg     <var1,var2,var3>
```

assumes

Syntax

assumes *segReg*, *segName*

This macro makes all references to data and code in the segment *segName* relative to the segment register given by *segReg*. It is similar to the **ASSUME** assembler directive.

The *segReg* parameter specifies the name of a segment register.

The *segName* parameter specifies the name of a predefined segment, **CODE** or **DATA**, or a user-defined segment.

Examples

```
assumes CS, CODE
assumes DS, CODE
```

cBegin

Syntax

cBegin [[*procName*]]

This macro defines the actual entry point for the function **procName**. The macro creates code that sets up the frame and saves registers.

The optional *procName* parameter specifies a function name. If it is given, it must be the same as the name given in the **cProc** macro immediately preceding the **cBegin** macro.

cCall

Syntax

cCall *procName*, [[<*argList*>]], [[<*underscores*>]]

This macro pushes the arguments in *argList* onto the stack, saves registers (if any), and calls the function *procName*.

The *procName* parameter specifies the name of the function to be called.

The optional *argList* parameter specifies a list of the names of arguments to be passed to the function. This list is not required if the **Arg** macro is used before **cCall.**

The optional *underscores* parameter specifies whether an underscore should be added to the beginning of *procName*. If this argument is blank and the calling convention is the C calling convention, an underscore is added.

Comments

The arguments of an **Arg** macro are pushed onto the stack before any arguments in the *argList* parameter of a **cCall** macro.

Byte-type parameters are passed as words. There is no sign extension or zeroing of the high-order byte.

Immediate arguments are not supported.

Examples

```
cCall   there,<pExt,ax,bx,pResult>
```

```
Arg     pExt
Arg     ax
cCall   there,<bx,pResult>
```

cEnd

Syntax

cEnd [[*procName*]]

This macro defines the exit point for the **procName** function. The macro creates code that discards the frame, restores registers, and returns to the caller.

The optional *procName* parameter specifies a function name. If it is given, it must be the same as the name given in the **cBegin** macro immediately preceding the **cEnd** macro.

Once a function has been defined using **cProc**, any formal parameters should be declared with the **parm***X* macro and any local variables with the **local***X* macro. The **cBegin** and **cEnd** macros must be used to delineate the code for the function.

The following is an example of a complete function definition:

Example

```
cProc   strcpy,<PUBLIC>,<si,di>
    parmW   dst
    parmW   src
    localW  cnt

cBegin
    cld
    mov     si,src
    mov     di,dest
    push    ds
    pop     es
    xor     cx,cx
    mov     cnt,cx
loop:
    lodsb
    stosb
    inc     cnt
    cmp     al,0
    jnz     loop
    mov     ax,cnt
cEnd
```

codeOFFSET

Syntax

codeOFFSET *arg*

This macro generates an offset relative to the start of the group to which the **CODE** segment belongs. It is similar to the **OFFSET** assembler operator, but automatically provides the group name. For this reason, it should be used instead of **OFFSET**.

The *arg* parameter specifies a label name or offset value.

Example

```
mov ax,codeOFFSET label
```

cProc

Syntax

cProc *procName*, *<attributes>*, *<autoSave>*

This macro defines the name and attributes of a function.

The *procName* parameter specifies the name of the function.

The *attributes* parameter specifies the function type. It can be a combination of the following:

Type	Description
NEAR	A near function. It can only be called from the segment in which it is defined.
FAR	A far function. It can be called from any segment.
PUBLIC	A public function. It can be externally declared in other source files.

The default attribute is **NEAR** and private (i.e., cannot be declared externally in other source files). The **NEAR** and **FAR** attributes cannot be used together. If more than one attribute is selected, the angle brackets are required.

The *autoSave* parameter specifies a list of registers to be saved when the function is invoked, and restored when exited. Any of the 8086's registers can be specified.

Comments

If this function is called by a function written in C, it *must* save and restore the SI and DI registers.

The BP register is always saved, regardless of whether it is present in the *autoSave* list.

Examples

```
cProc proc1, <FAR, ds,es>
cProc proc2, <NEAR,PUBLIC>
cProc proc3,,ds
```

createSeg

Syntax

createSeg *segName*, *logName*, *align*, *combine*, *class*

This macro creates a new segment that has the specified name and segment attributes. The macro automatically creates an **assumes** macro and an **OFFSET** macro for the new segment. This macro is intended to be used in medium-model Windows applications to define nonresident segments. The *segName* parameter specifies the actual name of the segment. This name is passed to the linker.

The *logName* parameter specifies the logical name of the segment. This name is used in all subsequent **sBegin**, **sEnd**, and **assumes** macros that refer to the segment.

The *align* parameter specifies the alignment type. It can be any one of the following:

BYTE
WORD
PARA
PAGE

The *combine* parameter specifies the combine type for the segment. It can be any one of the following:

COMMON
MEMORY
PUBLIC
STACK

If no combine type is given, a private segment is assumed.

The *class* parameter specifies the class name of the segment. The class name defines which segments must be loaded in consecutive memory.

Example

```
createSeg   _INIT,INITCODE,BYTE,PUBLIC,CODE

sBegin  INITCODE
assumes CS:INITCODE

       mov ax,initcodeOFFSET  sample

sEnd    INITCODE
```

Comments

The alignment, combine type, and class name are described in detail in the *Microsoft Macro Assembler Reference* .

The Cmacros have two predefined segments, named **CODE** and **DATA**, that any application can use without special definition. Medium-, large-, and huge-model applications can define additional segments by using the **createSeg** macro.

dataOFFSET

Syntax

dataOFFSET *arg*

This macro generates an offset relative to the start of the group to which the **DATA** segment belongs. It is similar to the **OFFSET** assembler operator, but automatically provides the group name. For this reason, it should be used instead of **OFFSET**.

The *arg* parameter specifies a label name or offset value.

Example

```
mv ax,dataOFFSET label
```

Def*X*

Syntax

Def*X* <*namelist*>

This macro registers the name of a user-defined variable with the Cmacros. Variables that are not defined using the **static***X*, **global***X*, **extern***X*, **parm***X*, or **local***X* macros cannot be referred to in other macros unless the name is registered, or the variable was defined with the **DW** assembler directive.

The *X* parameter specifies the storage size of the variable. It can be any one of the following:

Type	Description
B	Byte
W	Word
D	Double-word
Q	Quad-word
T	Ten-byte word
CP	Code pointer (one word for small and compact models)
DP	Data pointer (one word for small and medium models)

The *namelist* parameter specifies a list of variable names to be defined.

Example

```
maxSize db      132
        DefB    maxSize
dest equ  wordptr es:[di]
     DefW dest
```

errn$

Syntax

errn$ *label*, [[*bias*]]

This macro subtracts the offset of *label* from the offset of the location counter, then adds *bias* to the result. If this result is not zero, then an error message is displayed.

The *label* parameter specifies a label corresponding to a memory location.

The optional *bias* parameter specifies a signed bias value. A plus or minus sign is required.

Example

```
;   end of previous code
     errn$   function1
function1:
```

If a function that was originally located immediately after another piece of code is ever moved, **errn$** displays an error message.

errnz

Syntax

errnz <*expression*>

This macro evaluates a given expression. If the result is not zero, an error is displayed.

The *expression* parameter specifies the expression to be evaluated. The angle brackets are required if there are any spaces in the expression.

Examples

```
x       db      ?
y       db      ?

mov     ax, word ptr x
errnz   <(OFFSET y) - (OFFSET x) -1>
```

If during assembly, *x* and *y* receive anything but sequential storage locations, **errnz** displays an error message.

```
table1   struc
     .
     .
     .
table1len equ  $-table1
table1    ends

table2    struc
     .
     .
     .
table2len equ  $-table2
table2    ends

errnz   table1Len-table2Len
```

If during assembly, the length of two tables is not the same, **errnz** displays an error message.

extern*X*

Syntax

extern*X* <*namelist*>

This macro defines one or more names that will be the labels of external variables or functions.

The *X* parameter specifies the storage size or function type. It can be any one of the following:

Type	Description
A	Constant value declared with the EQU and = directives in a separate file
B	Byte
W	Word
D	Double-word
Q	Quad-word
T	Ten bytes
CP	Code pointer (one word for small and compact models)
DP	Data pointer (one word for small and medium models)
NP	Near function pointer
FP	Far function pointer
P	Near for small and compact models; far for other models

The *namelist* parameter specifies the list of the names of the variables or functions.

Examples

```
externB <DataBase>
externFP <SampleRead>
```

FarPtr

Syntax

FarPtr *name*, *segment*, *offset*

This macro defines a 32-bit pointer value that can be passed as a single argument in a **cCall** macro. In the **FarPtr** macro, the *segment* and *offset* values do not have to be in registers.

The *name* parameter specifies the name of the pointer to be created.

The *segment* parameter specifies the text that defines the segment portion of the pointer.

The *offset* parameter specifies the text that defines the offset portion of the pointer.

Example

```
FarPtr  destPtr,es,<wordptr 3[si]>
cCall   proc,<destPtr,ax>
```

globalX

Syntax

global*X* *name*, ⟦*initialValue*⟧ ⟦*replication*⟧

This macro allocates public static-memory storage.

The *X* parameter specifies the size of the storage to be allocated. It can be any one of the following:

Type	Description
B	Byte
W	Word
D	Double-word
Q	Quad-word
T	Ten bytes
CP	Code pointer (one word for small and compact models)
DP	Data pointer (one word for small and medium models)

The *name* parameter specifies the reference name of the allocated memory.

The optional *initialValue* parameter specifies an initial value for the storage. The default is zero if no value is specified.

The optional *replication* parameter specifies a count of the number of times the allocation is to be duplicated. This parameter generates the **DUP** assembler operator.

Examples

```
globalW  flag,1
globalB  string,0, 30
```

label*X*

Syntax

label*X* <*namelist*>

This macro defines one or more names that will be the labels of public (global) variables or functions.

The *X* parameter specifies the storage size or function type. It can be any one of the following:

Type	Description
B	Byte
W	Word
D	Double-word
Q	Quad-word
T	Ten bytes
CP	Code pointer (one word for small and compact models)
DP	Data pointer (one word for small and medium models)
NP	Near function pointer
FP	Far function pointer
P	Near for small and compact models; far for other models

The *namelist* parameter specifies the list of the names of the external variables or functions.

Examples

```
labelB <DataBase>
labelFP <SampleRead>
```

local*X*

Syntax

local*X* <*namelist*>, *size*

This macro defines one or more frame variables for the function. To keep the words in the stack aligned, the macro ensures that the total space allocated is an even number of bytes.

The *X* parameter specifies the storage size. It can be any one of the following:

Type	Description
B	Byte (allocates a single byte of storage on the stack)
W	Word (allocated on a word boundary)
D	Double-word (allocated on a word boundary)
V	Variable size (allocated on a word boundary)
Q	Quad-word (aligned on a word boundary)
T	Ten-byte word (aligned on a word boundary)
CP	Code pointer (one word for small and compact models)
DP	Data pointer (one word for small and medium models)

The *namelist* parameter specifies the list of the names of the frame variables for the function.

The *size* parameter specifies the size of the variable. It is used with **localV** only.

Comments

B-type variables are not necessarily aligned on word boundaries.

The **localD** macro creates two additional symbols, **OFF**_*name* and **SEG**_*name*. **OFF**_*name* is the offset portion of the parameter; **SEG**_*name* is the segment portion.

Only the name is required when referencing a variable. Write your code like this:

```
mov     al,var1
```

Not like this:

```
mov     al,byte ptr var1[bp]
```

Examples

```
localB <L1,L2,L3>
localW L4
localD <L5>
localV L6,%(size struc)
```

parm*X*

Syntax

parm*X* <*namelist*>

This macro defines one or more function parameters. The parameters provide access to the arguments passed to the function. Parameters must appear in the same order as the arguments in the function call.

The *X* parameter specifies the storage size. It can be any one of the following:

Type	Description
B	Byte (allocated on a word boundary on the stack)
W	Word (allocated on a word boundary)
D	Double-word (allocated on a word boundary)
Q	Quad-word (aligned on a word boundary)
T	Ten-byte word (aligned on a word boundary)
CP	Code pointer (one word for small and compact models)
DP	Data pointer (one word for small and medium models)

The *namelist* parameter specifies the list of the parameter names.

Comments

The **parmD** macro creates two additional symbols, **OFF**_*name* and **SEG**_*name*. **OFF**_*name* is the offset portion of the parameter; **SEG**_*name* is the segment portion.

Only the parameter name is required when referring to the corresponding argument. Write your code like this:

```
mov     al,var1
```

Not like this:

```
mov     al,byte ptr var1[bp]
```

Examples

```
parmW var1
parmB <var2,var3,var4>
parmD <var5>
```

Save

Syntax

Save *<regList>*

This macro directs the next **cCall** macro to save the specified registers on the stack before calling a function, and to restore the registers after the function returns. The macro can be used to save registers that are destroyed by the called function.

The **Save** macro applies to only one **cCall** macro; each new **cCall** must have a corresponding **Save** macro. If two **Save** macros appear before a **cCall**, only the second macro is recognized.

The *regList* parameter specifies a list of registers to be saved.

Examples

```
Save    <cl,bh,si>
Save    <ax>
```

sBegin

Syntax

sBegin *segName*

This macro opens up a segment. It is similar to the **SEGMENT** assembler directive.

The *segName* parameter specifies the name of the segment to be opened. It can be one of the predefined segments, **CODE** or **DATA**, or the name of a user-defined segment.

Examples

```
sBegin DATA
sBegin CODE
```

*segName*OFFSET

Syntax

*segName***OFFSET** *arg*

This macro generates an offset relative to the start of the group to which the user-defined segment *segName* belongs. It is similar to the **OFFSET** assembler operator, but automatically provides the group name. For this reason, it should be used instead of **OFFSET**.

The *arg* parameter specifies a label name or offset value.

Example

```
mv ax,initcodeOFFSET label
```

sEnd

Syntax

sEnd ⟦*segName*⟧

This macro closes a segment. It is similar to the **ENDS** assembler directive.

The optional *segName* parameter specifies a name used for readability. If it is given, it must be the same as the name given in the matching **sBegin** macro.

Examples

```
sEnd
sEnd DATA
```

static*X*

Syntax

static*X* *name*, ⟦*initialValue*⟧, ⟦*replication*⟧

This macro allocates private static-memory storage.

The *X* parameter specifies the size of storage to be allocated. It can be any one of the following:

Type	Description
B	Byte
W	Word
D	Double-word
Q	Quad-word
T	Ten bytes
CP	Code pointer (one word for small and compact models)
DP	Data pointer (one word for small and medium models)

The *name* parameter specifies the reference name of the allocated memory. The optional *initialValue* parameter specifies an initial value for the storage. If no value is specified, the default is zero.

The optional *replication* parameter specifies a count of the number of times the allocation is to be duplicated. This parameter generates the **DUP** assembler operator.

Examples

```
staticW  flag,1
staticB  string, , 30
```

Chapter 15

Windows DDE Protocol Definition

The Microsoft Windows Dynamic Data Exchange (DDE) protocol defines the method for communicating among applications. This communication takes place as applications send messages to each other to initiate conversations, to request and share data, and to terminate conversations. This chapter describes these messages and the rules associated with their use. It also briefly describes several clipboard formats which a DDE application can register for use in a DDE conversation.

Guide to Programming provides an overview of DDE programming, including such concepts as client, server, application, topic and item. It also introduces the modes of DDE communication, including permanent data links, one-time transfers, and remote command execution, and it explains the flow of DDE messages.

Conventions Used in This Chapter

Message-specific argument names bear prefixes indicating their type, as follows:

Prefix	Description
a	An atom of word length (16 bits); for example, *aName*.
cf	A registered clipboard format number (word length); for example, *cfFormat*.
f	A flag bit; for example, *fName*.
h	A handle (word length) to a global memory object; for example, *hName*.
w	Any other word-length argument; for example, *wName*.

15.1 Using the DDE Message Set

Each DDE message has two parameters. The first parameter, *wParam* (word length), carries the handle of the sender's window; it is the same in all cases and so is not shown in Table 15.1. The second parameter, *lParam* (a long word, 32 bits), is composed of a low-order word and a high-order word containing message-specific arguments, as follows:

Table 15.1 DDE Messages

Message	Arguments in *lParam* Low-order word	High-order word
WM_DDE_ACK		
In reply to INITIATE	*aApplication*	*aTopic*
In reply to EXECUTE	*wStatus*	*hCommands*
All other messages	*wStatus*	*aItem*
WM_DDE_ADVISE	*hOptions*	*aItem*
WM_DDE_DATA	*hData*	*aItem*
WM_DDE_EXECUTE	(Reserved)	*hCommands*
WM_DDE_INITIATE	*aApplication*	*aTopic*
WM_DDE_POKE	*hData*	*aItem*
WM_DDE_REQUEST	*cfFormat*	*aItem*
WM_DDE_TERMINATE	(Reserved)	(Reserved)
WM_DDE_UNADVISE	(Reserved)	*aItem*

An application calls the **SendMessage** function to issue the WM_DDE_INITIATE message or a WM_DDE_ACK message sent in response to WM_DDE_INITIATE. All other messages are sent using the **PostMessage** function. The window handle of the receiving window appears as the first parameter of these calls. The second parameter contains the message to be sent, the third parameter identifies the sending window, and the fourth parameter contains the message-specific arguments. For example:

```
PostMessage(hwndRecipient, WM_DDE_MESSAGE, hwndSender,
    MAKELONG(low_word, high_word))
```

The MAKELONG macro combines low_word and high_word into a long word.

15.2 Synchronizing the DDE Conversation

An application window that processes DDE requests from the window of a DDE partner must process them strictly in the order in which they are received from that partner. However, when handling messages from multiple DDE partners, the window does not have to follow this "first in, first out" rule. In other words, only the conversations themselves must be synchronous; the window can shift from one conversation to another asynchronously.

For example, suppose the following messages are in a window's queue:

Message from window X
Message from window Y
Message from window X

The window must process message 1 before message 3, but it need not process message 2 before message 3. If window Y is a lower-priority DDE-conversation partner than window X, the window can defer processing the messages from window Y until it has finished dealing with the messages sent by window X. The following shows acceptable processing orders for these messages and the relative priority implied by each order:

Order	Relative Priority
1 2 3	Window X = window Y
1 3 2	Window X > window Y
2 1 3	Window X < window Y

If an application is unable to process an incoming request because it is waiting for a DDE response, it must post a WM_DDE_ACK message with the **fBusy** flag set to 1 to prevent deadlock. An application can also send a busy WM_DDE_ACK message if for any reason the application cannot process an incoming request within a reasonable amount of time.

An application should be able to deal with the situation in which its DDE partner fails to respond with a message within a certain time-out interval. Since the length of this interval may vary depending on the nature of the application and the configuration of the user's system (including whether it is on a network), the application should provide a way for the user to specify the time-out interval.

15.3 Using Atoms

Certain arguments of DDE messages (*aItem*, *aTopic*, and *aApplication*) are global atoms. Applications using these atoms must explicitly delete them to purge them from the atom list. Section 15.7, "DDE Message Directory," describes the rules for allocating and deleting atoms used by each message.

In all cases, the sender of a message must delete any atom which the intended receiver will not receive due to an error condition, such as failure of the **PostMessage** function.

15.4 Using Shared Memory Objects

DDE uses shared memory objects for three purposes:

- To carry a data item value to be exchanged. This is an item referenced by the *hData* argument in the WM_DDE_DATA and WM_DDE_POKE messages.
- To carry options in a message. This is an item referenced by the *hOptions* argument in a WM_DDE_ADVISE message.
- To carry an execution-command string. This is an item referenced by the *hCommands* argument in the WM_DDE_EXECUTE message and its corresponding WM_DDE_ACK message.

Applications that receive a DDE shared memory object must treat it as read only. It must not be used as a mutual read/write area for the free exchange of data.

As with a DDE atom, a shared memory object should be freed properly to provide for effective memory management. Shared memory objects should be properly locked and unlocked. Section 15.7, "DDE Message Directory," describes the rules for allocating and deleting shared memory objects used by each message.

In all cases, the sender of a message must delete any shared memory object which the intended receiver will not receive due to an error condition, such as failure of the **PostMessage** function.

15.5 Using Clipboard Formats

You can pass data by means of any of the standard clipboard formats or with a registered clipboard formats. See the description of the **SetClipboardData** function in Chapter 4, "Functions Directory," in *Reference, Volume 1*, for more information on standard clipboards. See the description of the **RegisterClipboardFormat** function for information on registering clipboard formats.

A special, registered format named Link is used to identify an item in a DDE conversation. For more information, see *Guide to Programming*.

15.6 Using the System Topic

Applications are encouraged to support at all times a special topic with the name System. This topic provides a context for items of information that may be of general interest to another application.

The following list contains suggested items for the System topic. This list is not exclusive. The data item values should be rendered in the CF_TEXT format. Individual elements of a System topic item value should be delimited by tab characters.

Item	Description
SysItems	A list of the System-topic items supported by the application.
Topics	A list of the topics supported by the application at the current time; this list can vary from moment to moment.
ReturnMessage	Supporting detail for the most recently used WM_DDE_ACK message. This is useful when more than eight bits of application-specific return data are required.
Status	An indication of the current status of the application. When a server receives a WM_DDE_REQUEST message for this System-topic item, it should respond by posting a WM_DDE_DATA message with a string containing either "Busy" or "Ready," as appropriate.
Formats	A list of clipboard format numbers that the application can render.

15.7 DDE Message Directory

This section describes the nine DDE messages. Included in each description is a list of the message-specific arguments and the rules for posting and receiving each message. The SDK contains the DDE.H header file which defines the DDE messages and data structures described in this section.

WM_DDE_ACK

This message notifies an application of the receipt and processing of a WM_DDE_INITIATE, WM_DDE_EXECUTE, WM_DDE_DATA, WM_DDE_ADVISE, WM_DDE_UNADVISE, or WM_DDE_POKE message, and in some cases, of a WM_DDE_REQUEST message.

Parameter	Description
wParam	Identifies the sending window.
lParam	The meaning of the low-order and high-order words depends on the message to which the WM_DDE_ACK message is responding.

When responding to WM_DDE_INITIATE:

Argument	Description
aApplication	Low-order word of *lParam*. An atom that contains the name of the replying application.
aTopic	High-order word of *lParam*. An atom that contains the topic with which the replying server window is associated.

When responding to WM_DDE_EXECUTE:

Argument	Description
wStatus	Low-order word of *lParam*. A series of flags that indicate the status of the response.
hCommands	High-order word of *lParam*. A handle that identifies the data item containing the command string.

Parameter	Description
	When replying to all other messages:

Argument	Description
wStatus	Low-order word of *lParam*. A series of flags that indicate the status of the response.
aItem	High-order word of *lParam*. An atom that specifies the data item for which the response is sent.

Comments

The *wStatus* word consists of a **DDEACK** data structure that contains the following information:

Bit	Name	Meaning
15	**fAck**	1 = Request accepted.
		0 = Request not accepted.
14	**fBusy**	1 = Busy. An application is expected to set **fBusy** if it is unable to respond to the request at the time it is received. The **fBusy** flag is defined only when **fAck** is zero.
		0 = Not busy.
13–8		Reserved for Microsoft use.
7–0	**bAppReturnCode**	Reserved for application-specific return codes.

Posting

Except in response to the WM_DDE_INITIATE message, post the WM_DDE_ACK message by calling the **PostMessage** function, not **SendMessage**. When responding to WM_DDE_INITIATE, send the WM_DDE_ACK message with **SendMessage**.

When acknowledging any message with an accompanying *aItem* atom, the application that sends WM_DDE_ACK can reuse the *aItem* atom that accompanied the original message, or it may delete it and create a new one.

When acknowledging WM_DDE_EXECUTE, the application that sends WM_DDE_ACK should reuse the *hCommands* object that accompanied the original WM_DDE_EXECUTE message.

If an application has initiated the termination of a conversation by sending WM_DDE_TERMINATE and is awaiting confirmation, the waiting application should not acknowledge (positively or negatively) any subsequent message sent by the other application. The waiting application should delete any atoms or shared memory objects received in these intervening messages.

Receiving The application that receives WM_DDE_ACK should delete all atoms accompanying the message.

If the application receives WM_DDE_ACK in response to a message with an accompanying *hData* object, the application should delete the *hData* object.

If the application receives a negative WM_DDE_ACK message sent in reply to a WM_DDE_ADVISE message, the application should delete the *hOptions* object sent with the original WM_DDE_ADVISE message.

If the application receives a negative WM_DDE_ACK message sent in reply to a . WM_DDE_EXECUTE message, the application should delete the *hCommands* object sent with the original WM_DDE_EXECUTE message.

WM_DDE_ADVISE

This message, posted by a client application, requests the receiving (server) application to supply an update for a data item whenever it changes.

Parameter	Description
wParam	Identifies the sending window.
lParam	Identifies the requested data and specifies how the data is to be sent.

Argument	Description
hOptions	Low-order word of *lParam*. A handle to a global memory object that specifies how the data is to be sent.
aItem	High-order word of *lParam*. An atom that specifies the data item being requested.

Comments The global memory object identified by *hOptions* consists of a **DDEADVISE** data structure that contains the following:

Word	Name	Content
1	**fAckReq**	If bit 15 is 1, the receiving (server) application is requested to send its WM_DDE_DATA messages with the ACK-requested bit (**fAckReq**) set. This offers a flow-control technique whereby the client application can avoid overload from incoming DATA messages.
	fDeferUpd	If bit 14 is 1, the server is requested to send its WM_DDE_DATA messages with a null *hData* handle. These messages are alarms telling the client that the source data has changed. Upon receiving one of these alarms, the client can choose to call for the latest version of the data by issuing a WM_DDE_REQUEST message, or it can choose to ignore the alarm altogether. This would typically be used when there is a substantial resource cost associated with rendering and/or assimilating the data.
	reserved	Bits 13–0 are reserved.
2	**cfFormat**	The client's preferred type of data. Must be a standard or registered clipboard data format number.

If an application supports more than one clipboard format for a single topic and item, it can post multiple WM_DDE_ADVISE messages for the topic and item, specifying a different clipboard format with each message.

Posting

Post the WM_DDE_ADVISE message by calling the **PostMessage** function, not **SendMessage.**

Allocate *hOptions* by calling the **GlobalAlloc** function with the GEMEM_DDE_SHARE option.

Allocate *aItem* by calling the **GlobalAddAtom** function.

If the receiving (server) application responds with a negative WM_DDE_ACK message, the sending (client) application must delete the *hOptions* object.

Receiving

Post the WM_DDE_ACK message to respond positively or negatively. When posting WM_DDE_ACK, reuse the *aItem* atom or delete it and create a new one. If the WM_DDE_ACK message is positive, delete the *hOptions* object; otherwise, do not delete the object.

WM_DDE_DATA

This message, posted by a server application, sends a data item value to the receiving (client) application, or notifies the client of the availability of data.

Parameter	Description
wParam	Identifies the sending window.
lParam	Identifies the available data and specifies how it is sent.

Argument	Description
hData	Low-order word of *lParam*. A handle that identifies the global memory object containing the data and additional information. The handle should be set to NULL if the server is notifying the client that the data item value has changed during a "warm link." A warm link is established by the client sending a WM_DDE_ADVISE message with the *fDeferUpd* bit set.
aItem	High-order word of *lParam*. An atom that identifies the data item for which data or notification is sent.

Comments The global memory object identified by *hData* consists of a **DDEDATA** data structure that contains the following:

Word	Name	Content
1	**fAckReq**	If bit 15 is 1, the receiving (client) application is expected to send a WM_DDE_ACK message after the WM_DDE_DATA message has been processed. If bit 15 is zero, the client application should not send a WM_DDE_ACK message.
	reserved	Bit 14 is reserved.

Word	Name	Content
	fRelease	If bit 13 is 1, the client application is expected to free the *hData* memory object after processing it. If bit 13 is zero, the client application should not free the object. See the "Posting" and "Receiving" sections for exceptions.
	fRequested	If bit 12 is 1, this data is offered in response to a WM_DDE_REQUEST message. If bit 12 is zero, this data is offered in response to a WM_DDE_ADVISE message.
	reserved	Bits 11–0 are reserved.
2	**cfFormat**	This specifies the format in which the data are sent or offered to the client application. It must be a standard or registered clipboard data format.
3–*n*	**Value[ ]**	This is the data. It is in the format specified by **cfFormat**.

Posting

Post the WM_DDE_DATA message by calling the **PostMessage** function, not **SendMessage**.

Allocate *hData* by calling the **GlobalAlloc** function with the GMEM_DDESHARE option.

Allocate *aItem* by calling the **GlobalAddAtom** function.

If the receiving (client) application responds with a negative WM_DDE_ACK message, the sending (server) application must delete the *hData* object.

If the sending (server) application sets the **fRelease** flag to zero, the sender is responsible for deleting *hData* upon receipt of either a positive or negative acknowledgement.

Do not set both the **fAckReq** and **fRelease** flags to zero. If both flags are set to zero, it is difficult for the sending (server) application to determine when to delete *hData*.

Receiving

If **fAckReq** is 1, post the WM_DDE_ACK message to respond positively or negatively. When posting WM_DDE_ACK, reuse the *aItem* atom or delete it and create a new one.

If **fAckReq** is zero, delete the *aItem* atom.

If the sending (server) application specified *hData* as NULL, the receiving (client) application can request the server to send the actual data by posting a WM_DDE_REQUEST message.

After processing the WM_DDE_DATA message in which *hData* is not NULL, delete *hData* unless either of the following conditions is true:

- The **fRelease** flag is zero.
- The **fRelease** flag is 1, but the receiving (client) application responds with a negative WM_DDE_ACK message.

WM_DDE_EXECUTE

This message, posted by a client application, sends a string to a server application to be processed as a series of commands. The server application is expected to post a WM_DDE_ACK message in response.

Parameter	Description
wParam	Identifies the sending window.
lParam	Specifies the commands to be executed.

Argument	Description
reserved	The low-order word of *lParam* is reserved.
hCommands	High-order word of *lParam*. A handle that identifies a global memory object containing the command(s) to be executed.

Comments

The command string is null-terminated. The command string should adhere to the syntax shown below. Optional syntax elements are enclosed in double brackets (⟦ ⟧); single brackets ([ ]) are a syntax element.

[*opcodestring*] ⟦ [*opcodestring*] ⟧ ...

The *opcodestring* uses the following syntax:

opcode⟦ (*parameter* ⟦ ,*parameter* ⟧ ...) ⟧

The *opcode* is any application-defined single token. It may not include spaces, commas, parentheses, or quotation marks.

The *parameter* is any application-defined value. Multiple parameters are separated by commas, and the entire parameter list is enclosed in parentheses. The parameter may not include commas or parentheses except inside a quoted string. If a bracket or parenthesis character is to appear in a quoted string, it must be doubled: ((.

The following examples show valid command strings:

```
[connect][download(query1,results.txt)][disconnect]
[query("sales per employee for each district")]
[open("sample.xlm")][run("r1c1")]
```

Posting

Post the WM_DDE_EXECUTE message by calling the **PostMessage** function, not **SendMessage.**

Allocate *hCommands* by calling the **GlobalAlloc** function with the GMEM_DDE_SHARE option.

When processing WM_DDE_ACK sent in reply to WM_DDE_EXECUTE, the sender of the original WM_DDE_EXECUTE message must delete the *hCommands* object sent back in the WM_DDE_ACK message.

Receiving

Post the WM_DDE_ACK message to respond positively or negatively, reusing the *hCommands* object.

WM_DDE_INITIATE

This message, sent by either a client or server application, initiates a conversation with applications responding to the specified application and topic names.

Upon receiving this message, all applications with names that match the *aApplication* application and that support the *aTopic* topic are expected to acknowledge it (see the WM_DDE_ACK message).

Parameter	Description
wParam	Identifies the sending window.
lParam	Specifies the target application and the topic.

Argument	Description
aApplication	Low-order word of *lParam*. An atom that specifies the name of the application with which a conversation is requested. The application name may not contain slashes or backslashes. These characters are reserved for future use in network implementations. If the application name is NULL, a conversation with all applications is requested.

Parameter	Description	
	Argument	**Description**
	aTopic	High-order word of *lParam*. An atom that specifies the topic for which a conversation is requested. If the topic is NULL, a conversation for all available topics is requested.

Comments If the *aApplication* argument is NULL, any application may respond. If the *aTopic* argument is NULL, any topic is valid. Upon receiving a WM_DDE_INITIATE request with a null topic, an application is expected to send a WM_DDE_ACK message for each of the topics it supports.

Sending Send the WM_DDE_INITIATE message by calling the **SendMessage** function, not the **PostMessage** function. Broadcast the message to all windows by setting the first parameter of **SendMessage** to –1, as shown:

```
SendMessage(-1,WM_DDE_INITIATE,hwndClient,MAKELONG(aApp,aTopic));
```

If the application has already obtained the window handle of the desired server, it can send WM_DDE_INITIATE directly to the server window by passing the server's window handle as the first parameter of **SendMessage**.

Allocate *aApplication* and *aTopic* by calling **GlobalAddAtom**.

When **SendMessage** returns, delete the *aApplication* and *aTopic* atoms.

Receiving To complete the initiation of a conversation, respond with one or more WM_DDE_ACK messages, where each message is for a separate topic. When sending WM_DDE_ACK message, create new *aApplication* and *aTopic* atoms; do not reuse the atoms sent with the WM_DDE_INITIATE message.

WM_DDE_POKE

This message, posted by a client application, requests the receiving (server) application to accept an unsolicited data item value.

The receiving application is expected to reply with a positive WM_DDE_ACK message if it accepts the data, or with a negative WM_DDE_ACK message if it does not.

Parameter	Description
wParam	Identifies the sending window.
lParam	Identifies the data and specifies how it is sent.

Argument	Description
hData	Low-order word of *lParam*. A handle that specifies the global memory object containing the data and other information.
aItem	High-order word of *lParam*. An atom that identifies the data item offered to the server application.

Comments

The global memory object identified by *hData* consists of a **DDEPOKE** data structure that contains the following:

Word	Name	Content
1	reserved	Bits 15–14 are reserved.
	fRelease	If bit 13 is 1, the receiving (server) application is expected to free the memory object after processing it. If bit 13 is zero, the receiving application should not free the object. See the following "Posting" and "Receiving" sections for exceptions.
	reserved	Bits 12–0 are reserved.
2	**cfFormat**	This specifies the client's preferred type of data. It must be a standard or registered clipboard data format.
3–*n*	**Value[ ]**	This is the data. It is in the format specified by **cfFormat**.

Posting

Post the WM_DDE_POKE message by calling the **PostMessage** function, not **SendMessage.**

Allocate *hData* by calling the **GlobalAlloc** function with the GMEM_DDESHARE option.

Allocate *aItem* by calling the **GlobalAddAtom** function.

If the receiving (server) application responds with a negative WM_DDE_ACK message, the sending (client) application must delete the *hData* object.

If the sending (client) application sets the **fRelease** flag to zero, the sending application must delete *hData* upon receiving either a positive or negative WM_DDE_ACK message.

Receiving Post the WM_DDE_ACK message to respond positively or negatively. When posting WM_DDE_ACK, reuse the *aItem* atom or delete it and create a new one.

After processing the WM_DDE_POKE message, delete *hData* unless either of the following conditions is true:

- The **fRelease** flag is zero.
- The **fRelease** flag is 1, but the receiving (server) application responds with a negative WM_DDE_ACK message.

WM_DDE_REQUEST

This message, posted by a client application, requests the receiving (server) application to provide the value of a data item.

Parameter	Description
wParam	Identifies the sending window.
lParam	Specifies the requested data and the clipboard format number for the data

Argument	Description
cfFormat	Low-order word of *lParam*. A standard or registered clipboard format number.
aItem	High-order word of *lParam*. An atom that specifies which data item is being requested from the server.

Posting Post the WM_DDE_REQUEST message by calling the **PostMessage** function, not **SendMessage.**

Allocate **aItem** by calling the **GlobalAddAtom** function.

Receiving

If the receiving (server) application can satisfy the request, it responds with a WM_DDE_DATA message containing the requested data. Otherwise, it responds with a negative WM_DDE_ACK message.

When responding with either a WM_DDE_DATA or WM_DDE_ACK message, reuse the *aItem* atom or delete it and create a new one.

WM_DDE_TERMINATE

This message, posted by either a client or server application, terminates a conversation.

Parameter	Description
wParam	Identifies the sending window.
lParam	Is reserved.

Posting

Post the WM_DDE_TERMINATE message by calling the **PostMessage** function, not **SendMessage**.

While waiting for confirmation of the termination, the sending application should not acknowledge any other messages sent by the receiving application. If the sending application receives messages (other than WM_DDE_TERMINATE) from the receiving application, it should delete any atoms or shared memory objects accompanying the messages.

Receiving

Respond by posting a WM_DDE_TERMINATE message.

WM_DDE_UNADVISE

This message, sent by a client application, informs a server application that the specified item, or a particular clipboard format for the item, should no longer be updated. This terminates the warm or hot link for the specified item.

Parameter	Description
wParam	Identifies the sending window.
lParam	Specifies the data-request item to be canceled.

Parameter	Description	
	Argument	**Description**
	aItem	High-order word of *lParam*. An atom that specifies the data for which the update request is being retracted. When *aItem* is NULL, all active WM_DDE_ADVISE conversations associated with the client are to be terminated.
	cfFormat	Low-order word of *lParam*. The clipboard format of the item that specifies the clipboard format for which the update request is being retracted. When *cfFormat* is NULL, all active WM_DDE_ADVISE conversations for the item are to be terminated.

Posting

Post the WM_DDE_UNADVISE message by calling the **PostMessage** function, not **SendMessage.**

Allocate *aItem* by calling the **GlobalAddAtom** function.

Receiving

Post the WM_DDE_ACK message to respond positively or negatively. When posting WM_DDE_ACK, reuse the *aItem* atom or delete it and create a new one.

Appendixes

Appendix A
Virtual-Key Codes

The following list shows the symbolic constant names, hexadecimal values, and descriptive information for Microsoft Windows virtual-key codes. The codes are listed in numeric order.

Name	Value	Description
VK_LBUTTON	01H	Left mouse button
VK_RBUTTON	02H	Right mouse button
VK_CANCEL	03H	Used for control-break processing
VK_MBUTTON	04H	Middle mouse button (3-button mouse)
	05H–07H	Undefined
VK_BACK	08H	BACKSPACE key
VK_TAB	09H	TAB key
	0AH–0BH	Undefined
VK_CLEAR	0CH	CLEAR key
VK_RETURN	0DH	RETURN key
VK_SHIFT	10H	SHIFT key
VK_CONTROL	11H	CONTROL key
VK_MENU	12H	MENU key
VK_PAUSE	13H	PAUSE key
VK_CAPITAL	14H	CAPITAL key
	15H–19H	Reserved for Kanji systems
	1AH	Undefined
VK_ESCAPE	1BH	ESCAPE key
	1CH–1FH	Reserved for Kanji systems
VK_SPACE	20H	SPACEBAR
VK_PRIOR	21H	PAGE UP key
VK_NEXT	22H	PAGE DOWN key
VK_END	23H	END key
VK_HOME	24H	HOME key
VK_LEFT	25H	LEFT ARROW key
VK_UP	26H	UP ARROW key

Name	Value	Description
VK_RIGHT	27H	RIGHT ARROW key
VK_DOWN	28H	DOWN ARROW key
VK_SELECT	29H	SELECT key
	2AH	OEM specific
VK_EXECUTE	2BH	EXECUTE key
VK_SNAPSHOT	2CH	PRINTSCREEN key for Windows version 3.0 and later
VK_INSERT	2DH	INSERT key
VK_DELETE	2EH	DELETE key
VK_HELP	2FH	HELP key
VK_0	30H	0 key
VK_1	31H	1 key
VK_2	32H	2 key
VK_3	33H	3 key
VK_4	34H	4 key
VK_5	35H	5 key
VK_6	36H	6 key
VK_7	37H	7 key
VK_8	38H	8 key
VK_9	39H	9 key
	3AH–40H	Undefined
VK_A	41H	A key
VK_B	42H	B key
VK_C	43H	C key
VK_D	44H	D key
VK_E	45H	E key
VK_F	46H	F key
VK_G	47H	G key
VK_H	48H	H key
VK_I	49H	I key
VK_J	4AH	J key
VK_K	4BH	K key
VK_L	4CH	L key
VK_M	4DH	M key
VK_N	4EH	N key
VK_O	4FH	O key

Name	Value	Description
VK_P	50H	P key
VK_Q	51H	Q key
VK_R	52H	R key
VK_S	53H	S key
VK_T	54H	T key
VK_U	55H	U key
VK_V	56H	V key
VK_W	57H	W key
VK_X	58H	X key
VK_Y	59H	Y key
VK_Z	5AH	Z key
	5BH–5FH	Undefined
VK_NUMPAD0	60H	Numeric key pad 0 key
VK_NUMPAD1	61H	Numeric key pad 1 key
VK_NUMPAD2	62H	Numeric key pad 2 key
VK_NUMPAD3	63H	Numeric key pad 3 key
VK_NUMPAD4	64H	Numeric key pad 4 key
VK_NUMPAD5	65H	Numeric key pad 5 key
VK_NUMPAD6	66H	Numeric key pad 6 key
VK_NUMPAD7	67H	Numeric key pad 7 key
VK_NUMPAD8	68H	Numeric key pad 8 key
VK_NUMPAD9	69H	Numeric key pad 9 key
VK_MULTIPLY	6AH	Multiply key
VK_ADD	6BH	Add key
VK_SEPARATER	6CH	Separater key
VK_SUBTRACT	6DH	Subtract key
VK_DECIMAL	6EH	Decimal key
VK_DIVIDE	6FH	Divide key
VK_F1	70H	F1 key
VK_F2	71H	F2 key
VK_F3	72H	F3 key
VK_F4	73H	F4 key
VK_F5	74H	F5 key
VK_F6	75H	F6 key
VK_F7	76H	F7 key
VK_F8	77H	F8 key

Name	Value	Description
VK_F9	78H	F9 key
VK_F10	79H	F10 key
VK_F11	7AH	F11 key
VK_F12	7BH	F12 key
VK_F13	7CH	F13 key
VK_F14	7DH	F14 key
VK_F15	7EH	F15 key
VK_F16	7FH	F16 key
	80H–87H	OEM specific
	88H–8FH	Unassigned
VK_NUMLOCK	90H	NUM LOCK key
VK_OEM_SCROLL	91H	SCROLL LOCK key
	92H–B9H	Unassigned
VK_OEM_1	BAH	Keyboard-specific punctuation key (may not appear on every keyboard)
VK_OEM_PLUS	BBH	Plus (+) key
VK_OEM_COMMA	BCH	Comma (,) key
VK_OEM_MINUS	BDH	Minus (–) key
VK_OEM_PERIOD	BEH	Period (.) key
VK_OEM_2	BFH	Keyboard-specific punctuation key (may not appear on every keyboard)
VK_OEM_3	C0H	Keyboard-specific punctuation key (may not appear on every keyboard)
	C1H–DAH	Unassigned
VK_OEM_4	DBH	Keyboard-specific punctuation key (may not appear on every keyboard)
VK_OEM_5	DCH	Keyboard-specific punctuation key (may not appear on every keyboard)
VK_OEM_6	DDH	Keyboard-specific punctuation key (may not appear on every keyboard)
VK_OEM_7	DEH	Keyboard-specific punctuation key (may not appear on every keyboard)
VK_OEM_8	DFH	Keyboard-specific punctuation key (may not appear on every keyboard)
	E0H–E1H	OEM specific
VK_OEM_102	E2H	<> or \ | on enhanced, non-U.S. IBM®-compatible 102-key keyboard
	E3H–E4H	OEM specific

Name	Value	Description
	E5H	Unassigned
	E6H	OEM specific
	E7H–E8H	Unassigned
	E9H–F5H	OEM specific
	F6H–FEH	Unassigned

Appendix B
RC Diagnostic Messages

This appendix contains descriptions of diagnostic messages produced by the Resource Compiler (**RC**). Many of these messages appear when the Resource Compiler is not able to compile your resources. The descriptions in this appendix can help you determine the problem.

A (V) symbol at the beginning of a message description indicates that the message is displayed only if **RC** is run with the **–V** (verbose) option. These messages are generally informational and do not necessarily indicate errors.

See Chapter 8, "Rescource Script Statements," for information on the key words and fields specified in this appendix.

The messages are listed in alphabetical order.

Accelerator Type required (ASCII or VIRTKEY)

The *type* field in the **ACCELERATORS** statement must contain either the **ASCII** or **VIRTKEY** value.

BEGIN expected in Accelerator Table

The **BEGIN** key word must immediately follow the **ACCELERATORS** key word.

BEGIN expected in Dialog

The **BEGIN** key word must immediately follow the **DIALOG** key word.

BEGIN expected in menu

The **BEGIN** key word must immediately follow the **MENU** key word.

BEGIN expected in RCData

The **BEGIN** key word must immediately follow the **RCDATA** key word.

BEGIN keyword expected in String or Error Table

The **BEGIN** key word must immediately follow the **STRINGTABLE** or **ERRTABLE** key word.

Cannot Reuse String Constants

You are using the same value twice in a **STRINGTABLE** or **ERRTABLE** statement. Make sure you are not mixing overlapping decimal and hexadecimal values.

Control Character out of range [^A - ^Z]

A control character in the **ACCELERATORS** statement is invalid. The character following the caret (^) must be between A and Z, inclusive.

copy of *temp-file-2* **to** *exe-file* **failed**

The temporary file was not able to create the new .EXE file. Make sure that the TEMP environment variable is pointing to a drive that is not write-protected.

Copying segment *id* (*size* **bytes)**

(V) **RC** is copying the specified segment to the .EXE file.

Could not find RCPP.EXE

RCPP.ERR must be in the current directory or a directory in the PATH environment.

Could not open *in-file-name*

RC could not open the specified file. Make sure the file exists and that you typed the filename correctly.

Couldn't open *resource-name*

RC could not open the specified file. Make sure the file exists and that you typed the filename correctly.

Couldn't write executable

The .EXE file could not be copied to the temporary file. Make sure that the TEMP environment variable is pointing to a drive that is not write-protected and that the .EXE file from the linker is correct. You can check the .EXE file with the EXEHDR program.

Creating *resource-name*

(V) **RC** is creating a new .RES file.

Empty menus not allowed

An **END** key word appears before any menu items are defined in the **MENU** statement. Empty menus are not permitted by the Resource Compiler. Make sure you do not have any open quotation marks within the **MENU** statement.

END expected in Dialog

The **END** key word must occur at the end of a **DIALOG** statement. Make sure there are no open quotes left from the preceding statement.

END expected in menu

The **END** key word must come at the end of a **MENU** statement. Make sure you do not have any open quotation marks or a mismatched pair of **BEGIN** and **END** statements.

Error: Bitmap file *resource-file* **is not in 3.00 format.**

Use SDKPaint to convert version 2.x resource files to the 3.0 format.

Error Creating *resource-name*

Could not create specified .RES file. Make sure it is not being created on a read-only drive. Use the –V option to find out whether the file is being created.

Error: I/O error reading file.

Read failed. Since this is a generic routine, no specific filename is supplied.

Error: I/O error seeking in file

Seeking in file failed.

Error: I/O error writing file.

Write failed. Since this is a generic routine, no specific filename is supplied.

Error: Old DIB in *resource-name.* **Pass it through SDKPAINT.**

The resource file specified is not compatible with Windows 3.0. Make sure you have read and saved this file using the latest version of SDKPaint.

Error: Out of memory. Try not using resources with string identifiers.

There is not enough memory to allocate for a table of string names. You can view these names are when you use the **–V** option. Try to replace the string names with numbers. For example, you can change

```
MYICON  ICON myicon.ico
```

to

```
1       ICON myicon.ico
```

or provide the following statement in your header file:

```
#define MYICON 1
```

Error: Resource file *resouce-name* **is not in 3.00 format.**

Make sure your icons and cursors have been read and saved using the latest version of SDKPaint.

Errors in .EXE file

LINK failed. See the *CodeView and Utilities* manual in the Microsoft C 5.1 Optimizing Compiler documentation set for more information.

.EXE file too large; relink with higher /ALIGN value

The EXE file is too large. Relink the .EXE file with a larger **/ALIGN** value. If the .EXE file is larger than 800K, you should use the **/ALIGN:32** value on your **LINK** line.

.EXE not created by LINK

You must create the .EXE file with a version of **LINK** that is from C version 5.1 or later.

Expected Comma in Accelerator Table

RC requires a comma between the *event* and *idvalue* fields in the **ACCELERATORS** statement.

Expected control class name

The *class* field of a **CONTROL** statement in the **DIALOG** statement must be one of the following types: BUTTON, COMBOBOX, EDIT, LISTBOX, SCROLLBAR, STATIC, or user-defined. Make sure the class is spelled correctly.

Expected font face name

The *typeface* field of the **FONT** option in the **DIALOG** statement must be an ASCII character string enclosed in double quotation marks. This field specifies the name of a font.

Expected ID value for Menuitem

The **MENU** statement must contain a *menuID* field, which specifies the name or number that identifies the menu resource.

Expected Menu String

Each **MENUITEM** and **POPUP** statement must contain a *text* field, which is a string enclosed in double quotation marks that specifies the name of the menu item or pop-up menu. A **MENUITEM SEPARATOR** statement requires no quoted string.

Expected numeric command value

RC was expecting a numeric *idvalue* field in the **ACCELERATORS** statement. Make sure you have used a **#define** constant to specify the value and that the constant is spelled correctly.

Expected numeric constant in string table

A numeric constant, defined in a **#define** statement, must immediately follow the **BEGIN** key word in a **STRINGTABLE** or **ERRTABLE** statement.

Expected numeric point size

The *pointsize* field of the **FONT** option in the **DIALOG** statement must be an integer point size value.

Expected Numerical Dialog constant

A **DIALOG** statement requires integer values for the *x, y, width*, and *height* fields. Make sure these values are included after the **DIALOG** key word and that they are not negative.

Expected String in STRINGTABLE/ERRTABLE

A string is expected after each *stringid* value in a **STRINGTABLE** or **ERRTABLE** statement.

Expected String or Constant Accelerator command

RC was not able to determine what kind of key is being set up for the accelerator. The *event* field in the **ACCELERATORS** statement might be invalid.

Expecting number for ID

Expecting a number for the *id* field of a control statement in the **DIALOG** statement. Make sure you have a number or **#define** statement for the control ID.

Expecting quoted string in dialog class

The *class* field of the **CLASS** option in the **DIALOG** statement must be an integer or a string, enclosed in double quotation marks.

Expecting quoted string in dialog title

The *captiontext* field of the **CAPTION** option in the **DIALOG** statement must be an ASCII character string enclosed in double quotation marks.

File not found: *fileame*

The file specified in the **RC** command line was not found. Check to see whether the file has been moved to another directory and whether the filename or pathname is typed correctly.

Font names must be ordinals

The *pointsize* field in the **FONT** statement must be an integer, not a string.

Gangload area is [size] bytes at offset 0x[address]

(V) This is the size (in bytes) of all the segments that have one of the following attributes:

- **PRELOAD**
- **DISCARDABLE**
- Code segments that contain the entry point, WinMain
- Data segments (which should not be discardable)

The segments are placed in a contінguous area in the .EXE file for fast loading. The offset value is from the the beginning of the file. To disable gangloading, use the **–k** option.

Insufficient memory to spawn RCPP.EXE

There wasn't enough memory to run the preprocessor (RCPP). You can try not running any memory-resident software that might be taking up too much memory. Use the CHKDSK program to verify the amount of memory you have.

Invalid Accelerator

An *event* field in the **ACCELERATORS** statement was not recognized or was more than two characters in length.

Invalid Accelerator Type (ASCII or VIRTKEY)

The *type* field in the **ACCELERATORS** statement must contain either the **ASCII** or **VIRTKEY** value.

Invalid control character

A control character in the **ACCELERATORS** statement is invalid. A valid control character consists of one letter (only) following a caret (^).

Invalid Control type

Each control statement in a **DIALOG** statement must be one of the following: **CHECKBOX, COMBOBOX, CONTROL, CTEXT, DEFPUSHBUTTON, EDITTEXT, GROUPBOX, ICON, LISTBOX, LTEXT, PUSHBUTTON, RADIOBUTTON, RTEXT, SCROLLBAR.**

Make sure these control statements are spelled correctly.

Invalid .EXE file

The .EXE file is invalid. Make sure that the linker created it correctly and that the file exists. You can check the .EXE file with the EXEHDR program.

Invalid switch, *option*

You used an option that was not valid. Use **RC –?** for a list of the command-line options.

Invalid type

The resource type was not among the types defined in the WINDOWS.H file.

Invalid usage. Use rc –? for Help

Make sure you have at least one filename to work with. Use **RC –?** for a list of the command-line options.

No executable filename specified.

The **–FE** option was used, but no .EXE filename specified.

No resource binary filename specified.

The **–FO** option was used, but no .RES filename specified.

Not a Microsoft Windows format .EXE file

Make sure that the linker created the .EXE file correctly and that the file exists. You can check the .EXE file with the EXEHDR program.

Out of far heap memory

There wasn't enough memory. Try not running any memory-resident software that might be taking up too much space. Use the CHKDSK program to find out how much memory you have.

Out of memory, needed *n* **bytes**

RC was not able to allocate the specified amount of memory.

RC: Invalid swap area size: –S *string*

Invalid swap area size. Check your syntax for the **–S** option on the **RC** command line. The following are acceptable command lines:

```
RC  S123
RC  S123K  ;where K is kilobytes
RC  S123p  ;where p is paragraphs
```

RC: Invalid switch: *option*

You used an option that was not valid. Use **RC –?** for a list of the command-line options.

RC: RCPP *preprocessor-command-string*

(V) **RC** is passing the specified string to the preprocessor.

RC: RCPP.ERR not found

RCPP.ERR must be in the current directory or a directory in the PATH environment.

RC terminated by user

A CONTROL+C key combination was pressed, terminating **RC**.

RC terminating after preprocessor errors

See the Microsoft C 5.1 Optimizing Compiler documentation for information about preprocessor errors.

RCPP.EXE command line greater than 128 bytes

The command line was too long.

RCPP.EXE is not a valid executable

RCPP.EXE is not valid. The file might have been altered. Try copying the file from the SDK disks.

Reading *resource-name*

(V) **RC** is reading the .RES file.

Resources will be aligned on *number* **byte boundaries**

(V) The alignment is determined by the **ALIGN:***number* option on the **LINK** line.

Sorting preload segments and resources into gangload section

(V) **RC** is sorting the preloaded segments so that they can be loaded quickly.

Text string or ordinal expected in Control

The *text* field of a **CONTROL** statement in the **DIALOG** statement must be either a text string or an ordinal reference to the type of control is expected. If using an ordinal, make sure that you have a **#define** statement for the control.

The EXETYPE of this program is not Windows

The **EXETYPE WINDOWS** statement did not appear in the .DEF file. Since the linker might make optimizations for OS/2 (the default **EXETYPE**) that are not appropriate for Windows, the **EXETYPE WINDOWS** statement must be specified.

Unable to create *destination*

RC was not able to create the destination file. Make sure there is enough disk space.

Unable to open ***exe-file***

RC could not open this .EXE file. Make sure that the linker created it correctly and that the file exists.

Unbalanced Parentheses

Make sure you have closed every open parenthesis in the **DIALOG** statement.

Unexpected value in RCData

The *raw-data* values in the **RCDATA** statement must be integers or strings, each separated by a comma. Make sure you did not leave out a comma or leave out a quotation mark around a string.

Unknown DIB header format

The bitmap header is not a **BITMAPCOREHEADER** or **BITMAPINFO-HEADER** structure.

Unknown error spawning RCPP.EXE

For an unknown reason, RCPP was not started. Try copying the file from the SDK disks, and use the CHKDSK program to verify the amount of available memory.

Unknown Menu SubType

The *item-definition* field of the **MENU** statement can contain only **MENUITEM** and **POPUP** statements.

Warning: ASCII character not equivalent to virtual key code

There is an invalid virtual-key code in the **ACCELERATORS** statement. The ASCII value for some characters (such as *, ^, &,) is not equivalent to the virtual-key code for the corresponding key. (In the case of the asterisk (*), the virtual-key code is equivalent to the ASCII value for 8, the numeric character on the same key. Therefore the statement

```
VIRTKEY '* '
```

is invalid.) See Appendix A, "Virtual-Key Codes," and Appendix D, "Character Tables," for these values.

Warning: Discardable segment ***id*** **(*hex-size* bytes) is excessively large.**

The segment is greater than 27FFh in size. **RC** displays this warning because very large segments can adversely affect memory usage. Check your map file listing for the exact size of your segments.

Warning: SHIFT or CONTROL used without VIRTKEY

The **ALT**, **SHIFT**, and **CONTROL** options apply only to virtual keys in the **ACCELERATORS** statement. Make sure you have used the **VIRTKEY** option with one of these other options.

Writing resource *resource-name or ordinal-id resource type* (*resource size*)

(V) **RC** is writing the resource name or ordinal ID, followed by a period and the resource type and size (in bytes).

Warning: *string* **segment** *number* **set to PRELOAD**

RC displays this warning when it copies a segment that must be preloaded but that is not marked **PRELOAD** in the linker .DEF file.

All nondiscardable segments should be preloaded, including automatic data segments, fixed segments and the entry point of the code (WinMain).

The attributes of your code segments are set by the .DEF file. Check your map file listing for more information.

Appendix C
Windows Debugging Messages

The debugging version of Microsoft Windows generates diagnostic messages whenever it encounters an error that would otherwise cause the system to fail. Each diagnostic message has a unique number or string that identifies the cause of the message and potential failure. This appendix lists most of the diagnostic message names, their corresponding hexadecimal value, explains the meaning of each message, and in some cases suggests possible solutions.

The messages are divided into three sections that correspond to the three Windows modules: User, GDI, and Kernel. The messages in each section are produced by a function that is contained in the respective module. This division is necessary only because some messages in the User and GDI modules have the same error code.

User Error Codes

The error codes in this section are created by functions in the Windows User module. Some of these messages use the same codes as do GDI messages. Check the context of the error code to determine which module it is associated with. See the next section, "GDI Error Codes" for more information on differentiating these messages.

Code	Meaning
1	Not enough memory was available for the requested allocation. Ask for a smaller amount of memory. Check **HEAPWALK** to see how much memory is free. Make sure you are not creating fixed objects that are fragmenting memory.
2	Not enough memory was available for the requested reallocation. Do not attempt to call the **LocalRealloc** function to increase the size of your segment beyond 64K. Avoid creating fixed objects that fragment memory. Make sure that the object is not discardable.
4	The memory block could not be locked. Make sure the return value from your allocation function is a valid handle. Check **HEAPWALK** to see how much memory is free. Make sure you are not creating a fixed object that is fragmenting the memory.
5	The memory block could not be unlocked. Make sure the block was locked to begin with.

6	An invalid handle was passed to a GDI function. This could occur for any GDI object. Check the value you obtain from the Create/Get GDI object to make sure it returned a valid value.
7	The handle to the window you passed to the function was not valid. Use the **IsWindow** function to verify that the handle is valid and that the window has not been destroyed.
8	The five preallocated display contexts (DCs) are in use. Make sure your application calls the **ReleaseDC** function to release a DC when the application is done with it. If **ReleaseDC** is not called, the DC will not be available to the system or any application.
9	The **DefWindowProc** function was not found in your application. Place the **DefWindowProc** function in your application and make sure you are passing the correct parameters.
A	Some other application may have left the clipboard open. Pause and check again in a few seconds. Make sure your application calls the **CloseClipboard** function as soon as possible. Do not leave the clipboard open.
B	Your application attempted to destroy a window while it was still using a display context (DC). Make sure your application calls the **ReleaseDC** function to release a DC when the application is done with it. If **ReleaseDC** is not called, the DC will not be available to the system or any application.
C	The keyboard driver did not initialize correctly. Rerun Setup.
D	The mouse driver did not initialize correctly. Rerun Setup, or make sure that the mouse hardware did not get disconnected and that it is working outside of Windows.
E	The display driver did not initialize correctly. Rerun Setup.
F	An attempt was made to unlock the data segment but it wasn't locked. Make sure the data segment is locked before trying to unlock it.
16	The counter for windows of a particular class exceeded the limit of 32,767. Each time a window of a particular class is created, Windows increments a class usage counter. Each time a window of that class is destroyed, the counter is decremented. This message occurs in a **CreateWindow** or **CreateWindowEx** function.
17	The counter for windows of a particular class became a negative number. See the preceding message for details. This message occurs in a **DestroyWindow** function.

18	The counter for windows of a particular class was not zero when the class was destroyed. When an application or library terminates, Windows destroys all windows and classes created by that application or library. This error occurs if after the class is destroyed there still exists a window created by a different application or library that used the destroyed class.

GDI Error Codes

GDI errors occur when an invalid handle is passed to certain GDI functions. These errors can be identified by the existence of ValidateHandle in the back-trace. ValidateHandle is an internal Windows function that ensures that a handle is valid. Make sure you check for this function in order to differentiate GDI errors from User errors that have the same code number. (User messages are described in the previous section.)

Code	Meaning
0	A GDI function received a NULL object handle.
1 - A	A valid handle is referencing an object that is not a valid GDI object or is a GDI object of the wrong type. This error often occurs when an object is deleted and the handle is reused for some other purpose in another GDI operation.

The value of the error code depends on the type of object expected by the GDI function that generated the error. Each GDI object has a type identifier. Each GDI function that accepts an object as a parameter determines which object or objects are acceptable. To ensure that the handle it receives is valid, the GDI function calls ValidateHandle and passes it the handle and a range of acceptable type identifiers. If the handle references an object whose type identifier does not fall in the acceptable range, ValidateHandle generates an error code representing the lowest value of the range.

For example, the **SelectObject** expects its first parameter to be a DC, a metafile DC, or and banding metafile DC. It passes this value, along with the range (7H to AH) to ValidateHandle. If the type identifier of the handle is not within that range, ValidateHandle produces an error code with the value 7H.

The following list shows the type identifier of various objects:

Type ID	Object
1	Pen
2	Brush
3	Font
4	Palette

Type ID	Object
5	Bitmap
6	Region
7	Device context
8	Disabled device context
9	Metafile device context
A	Banding metafile device context

B	A window being destroyed had not released a DC that was obtained using the **GetDC** function.

Kernel Error Codes

The diagnostic messages in this section are associated with functions contained in the Windows Kernel module. These messages are listed in numerical order. Some numbers represent multiple messages. The retail version of Windows displays both the code number and the message text. The debugging version of Windows displays only the code number.

Code	Message
FF	**gnotify - can't discard segment** This error is usually caused in real mode by a far call when the DS register is pointing to a fixed object. Windows will not be able to discard the code segment that made the call. This error can be produced by the following functions: **GlobalReAlloc**, **GlobalAlloc** (the *wFlags* parameter cannot contain GMEM_NOCOMPACT or GMEM_NODISCARD), **GlobalCompact**, **GlobalDiscard**, **GlobalWire**.
FF	**Cannot GetProcAddress a task** You cannot use the **GetProcAddress** call for a library or the calling task.
FF	**MakeProcInstance only for current instance** This message is displayed if you use **MakeProcInstance** to call the entry point of another task.
FF	**MyOpenFile not reentrant** Internal Windows error.

FF	**gadd_free: Seg add not in range** Unable to add segment to global free list. Your application has stepped over Windows' memory.
FF	**FREE MEMORY OVERWRITE AT** Memory listed as free does not contain CC in each byte as expected. Put a break point at the specified address to find the problem.
FF	**free_list: prev bad** The free global memory list was corrupted by a wild write; the pointer from the previous entry in list does not point to the current entry.
FF	**free_list: next bad** The free global memory list was corrupted by a wild write; the pointer in the next entry does not point back to current entry.
FF	**free_list: count bad** The free global memory list was corrupted by a wild write; the final entry in the list does not match expected final entry.
FF	**Heap frozen in INT 3F.** Internal Windows error.
FF	**LOCAL FREE MEMORY OVERWRITE AT** The memory listed as free does not contain CC in each byte as expected.
FF	**Automatic Data Segment larger than 64K.** STACK + HEAP + STATICS combined are greater than 64K. Change the module-definition (.DEF) file.
FF	**PatchCodeHandle, CORE DUMP FOLLOWS:** Internal Windows error.
FF	**lru: prev bad** The free global memory list was corrupted by a wild write; the pointer from the previous entry in the list does not point to current entry.
FF	**lru: next bad** The free global memory list was corrupted by a wild write; the pointer in the next entry does not point back to the current entry.

FF **lru: count bad**

The free global memory list was corrupted by a wild write; the final entry in the list does not match the expected final entry.

100 **LocalAlloc : Invalid local heap**

A wild write corrupted the local heap.

100 **lfreeadd : Invalid local heap**

Unable to add segment to the local free list. Your application has overwritten the local heap.

100 *function_name*: **Invalid local heap**

Lists the function at which the check is occurring (**LocalAlloc**, **LocalLock**, etc.) and generally indicates an overwrite of the local heap list.

103 **Invalid local heap**

Either a wild write occurred or the **LocalInit** function was not performed correctly. Leave some memory for Windows when calling **LocalInit**.

140 **Local heap is busy**

Two edit controls in the same dialog with the same ID value. Make sure that you don't interchange decimal and hexadecimal numbers.

140 **EnterCrit: local heap is busy**

Internal Windows error. Attempting to reenter critical section of local memory manager.

140 **LeaveCrit: local heap is NOT busy**

Internal Windows error. Attempting to leave critical section of local memory manager when not already in critical section.

143 **Invalid local heap**

14B **Invalid local heap**

15B **Invalid local heap**

180 **LDREF: Invalid local handle**

A local handle (produced in calls to **LocalReAlloc**, **LocalLock**, etc.) is invalid.

1C0 **LocalLock: Object usage count overflow**

LMEM_MOVEABLE or LMEM_DISCARDABLE memory was locked more than the limit of 255 times.

1F0 **LocalFree: freeing locked object**

Local memory was not unlocked before **LocalFree** was called.

1F0 **LocalUnlock: Object usage count underflow**

Local memory was unlocked more times than it was locked.

200 **gmove_stack usage error**

Internal Windows error using temporary stack.

200 **Leave_eems_stack error**

Internal Windows error using temporary stack.

200 *function_name*: **Invalid global heap,***offender_para_reader_header*

Lists the function at which check is occurring (&n = **GlobalAlloc**, **GlobalLock**, etc.) and generally indicates overwrite of local heap list.

200 *function_name*: **Invalid global heap,***offender_para_reader_header*

Lists the function where check failed and generally indicates overwrite of local heap list.

If DX is nonzero, DX = offending arena header:

Code	Meaning
201	Forward links invalid
202	Backward links invalid
204	ga_handle points to free handle
208	Arena points to handle
280	ga_sig is bad

If DX is 0:

Code	Meaning
210	Allocated handles don't match used handles
220	Total number of handles don't match up
240	Total number of free handles does not match up

240 **Critical section problems**

280	**gdref: invalid handle**	

280 **gdref: invalid handle**

Global handle (produced in calls to **GlobalReAlloc**, **GlobalLock**, etc.) is invalid. Make sure you:

- Have a window procedure for the window.
- Do a **MakeProcInstance** call for the window procedure.
- Export your window procedure.

2C0 **GlobalLock: Object usage count overflow**

GMEM_MOVEABLE or GMEM_DISCARDABLE memory was locked more than the limit of 255 times.

2F0 **EMS_GlobalFree: freeing locked object**

Memory was not unlocked before **GlobalFree** was called.

2F0 **GlobalFree: freeing locked object**

Global memory was not unlocked before **GlobalFree** was called.

2F0 **GlobalFree: freeing locked object**

Global memory was not unlocked before **GlobalFree** was called.

2F0 **GlobalUnlock: Object usage count underflow**

Global memory was unlocked more times than it was locked.

2F0 **GlobalUnWire: Object usage count underflow**

Global memory was unwired more times than it was wired.

303 **PatchStack - invalid BP chain**

Stack frame chain was invalid due to a wild write.

303 **SearchStack - invalid BP chain**

Stack frame chain was invalid due to a wild write.

401 **BOOT: unable to load** ***application***

LoadModule failed for the shell application.

401 **BOOT: Unable to find file** ***pathname***

File not found.

401 **BOOT: Invalid .EXE file** *pathname*

.EXE file format is invalid.

401 **BOOT: Unable to load** *pathname*

LoadModule failed for a library loaded during boot time. Pass a far pointer to the name of the module that could not be loaded.

403 **Invalid ordinal reference**

You have linked to a function that does not have an entry point in the version of Windows you are running. Check your .DEF file to make sure you are using the correct ordinal reference.

404 **Call to undefined dynlink entry point at** *entry-point*

A bad import table or wild write occurred over segment relocation table. This message is displayed when your application calls the ordinal entry point for a driver that no longer contains that ordinal.

405 **Invalid start procedure**

Bad EXE header.

406 **Invalid module handle**

Could not obtain EXE header for the specified module handle.

407 **Invalid relocation record in** *es,bx*

A wild write destroyed a relocation record.

408 **Error saving forward reference**

409 **Out of memory loading segment from** *hModule of segment location*

Insufficient memory was available for loading segments.

409 **I/O error reading segment contents from** *hModule of segment location*

Unable to read segment due to file open, read, or seek error.

409 **Segment contents invalid**

Checksum value did not match segment contents when loading a segment.

409 **Segment contents trashed**

A wild write has occurred on the specified segment.

Error 409 occurs when a code segment was changed after it was loaded; this is usually the result of a wild write.

Running in the protected-mode version of Windows will cause a general protection violation to occur on the code causing the violation.

Check to make sure your buffers are large enough for the operation. Also, run Shaker to see if the problem occurs more frequently.

410 **Error reading relocation records from**

Int 21 function 3F was unable to read off the disk, or the information read is incompatible with the information requested.

411 **Insert disk for specified file**

412 **Unable to load non-resident name table**

When attempting to load the nonresident name table, one of the following four possible errors occurred:

- **OpenFile** failed.
- Int 21 function 42 (seek) failed.
- Int 21 function 3F (load seg) failed.
- The table size is inconsistent with the contents.

4FF **INT 3F handler unable to load segment**

LoadSegment failed. You will receive an "Out of memory loading segment" message before you receive this message.

501 **Missing resource table**

502 **Bad resource type**

503 **Bad resource name**

504 **Bad resource file**

505 **Unable to read resource from *segment***

Int 21 function 3F was unable to read off disk or the information read is incompatible with the information requested.

505 **Error loading from resource file** *filename*

This error has one of these possible causes:

- The *hResInfo* parameter of **LoadResource** is NULL.
- A wild write has destroyed the module header.
- A wild write has destroyed the EXE table.
- The resource file does not contain the resource requested.

600 **Atom Manager errors**

A wild write has occurred.

700 **Input/Output package errors**

Appendix D
Character Tables

IBM PC Extended Character Set

128	Ç	144	É	160	á	176	■	192	■	208	■	224	■	240	■
129	ü	145	æ	161	í	177	■	193	■	209	■	225	ß	241	±
130	é	146	Æ	162	ó	178	■	194	■	210	■	226	■	242	■
131	â	147	ô	163	ú	179	■	195	■	211	■	227	¶	243	■
132	ä	148	ö	164	ñ	180	■	196	■	212	■	228	■	244	■
133	à	149	ò	165	Ñ	181	■	197	■	213	■	229	■	245	■
134	å	150	û	166	ª	182	■	198	■	214	■	230	µ	246	■
135	ç	151	ù	167	º	183	■	199	■	215	■	231	■	247	■
136	ê	152	ÿ	168	¿	184	■	200	■	216	■	232	■	248	°
137	ë	153	Ö	169	—	185	■	201	■	217	■	233	■	249	■
138	è	154	Ü	170	¬	186	■	202	■	218	■	234	■	250	■
139	ï	155	¢	171	½	187	■	203	■	219	■	235	■	251	■
140	î	156	£	172	¼	188	■	204	■	220	■	236	■	252	■
141	ì	157	¥	173	¡	189	■	205	■	221	■	237	■	253	²
142	Ä	158	■	174	«	190	■	206	■	222	■	238	■	254	··
143	Å	159	■	175	»	191	■	207	■	223	■	239	■	255	■

■ Indicates that this character is not supported by Windows.

ANSI Table

0	■	32		64	@	96	`	128	■	160		192	À	224	à
1	■	33	!	65	A	97	a	129	■	161	¡	193	Á	225	á
2	■	34	"	66	B	98	b	130	■	162	¢	194	Â	226	â
3	■	35	#	67	C	99	c	131	■	163	£	195	Ã	227	ã
4	■	36	$	68	D	100	d	132	■	164	¤	196	Ä	228	ä
5	■	37	%	69	E	101	e	133	■	165	¥	197	Å	229	å
6	■	38	&	70	F	102	f	134	■	166	¦	198	Æ	230	æ
7	■	39	'	71	G	103	g	135	■	167	§	199	Ç	231	ç
8	■	40	(	72	H	104	h	136	■	168	¨	200	È	232	è
9	■	41	)	73	I	105	i	137	■	169	©	201	É	233	é
10	■	42	*	74	J	106	j	138	■	170	ª	202	Ê	234	ê
11	■	43	+	75	K	107	k	139	■	171	«	203	Ë	235	ë
12	■	44	,	76	L	108	l	140	■	172	¬	204	Ì	236	ì
13	■	45	-	77	M	109	m	141	■	173	-	205	Í	237	í
14	■	46	.	78	N	110	n	142	■	174	®	206	Î	238	î
15	■	47	/	79	O	111	o	143	■	175	¯	207	Ï	239	ï
16	■	48	0	80	P	112	p	144	■	176	°	208	Ð	240	ð
17	■	49	1	81	Q	113	q	145	‘	177	±	209	Ñ	241	ñ
18	■	50	2	82	R	114	r	146	’	178	²	210	Ò	242	ò
19	■	51	3	83	S	115	s	147	■	179	³	211	Ó	243	ó
20	■	52	4	84	T	116	t	148	■	180	´	212	Ô	244	ô
21	■	53	5	85	U	117	u	149	■	181	µ	213	Õ	245	õ
22	■	54	6	86	V	118	v	150	■	182	¶	214	Ö	246	ö
23	■	55	7	87	W	119	w	151	■	183	·	215	×	247	÷
24	■	56	8	88	X	120	x	152	■	184	¸	216	Ø	248	ø
25	■	57	9	89	Y	121	y	153	■	185	¹	217	Ù	249	ù
26	■	58	:	90	Z	122	z	154	■	186	º	218	Ú	250	ú
27	■	59	;	91	[	123	{	155	■	187	»	219	Û	251	û
28	■	60	<	92	\	124	|	156	■	188	¼	220	Ü	252	ü
29	■	61	=	93	]	125	}	157	■	189	½	221	Ý	253	ý
30	■	62	>	94	^	126	~	158	■	190	¾	222	Þ	254	þ
31	■	63	?	95	_	127	■	159	■	191	¿	223	ß	255	ÿ

■ Indicates that this character is not supported by Windows.

Appendix E
Windows 32-Bit Memory Management DLL

One of the significant features of the Intel 80386 and 80486 microprocessors is the availability of 32-bit registers for the manipulation of code and data. Applications written to use these registers can avoid the segmented memory model of earlier CPUs and instead use a "flat" memory model in which memory is viewed as a single, contiguous block.

Although Microsoft Windows version 3.0 continues to adhere to a segmented memory model, Windows does provide a set of functions that allow an application to make use of the 32-bit capabilities of the 80386 and 80486 processors. These functions are available to an application through a dynamic-link library (DLL) named WINMEM32.DLL. This DLL is supplied as part of the SDK and is not part of the retail version of Windows. Consequently, if your application calls functions in WINMEM32.DLL, you must include WINMEM32.DLL with your application when you distribute it to your application's end users.

This appendix introduces the functions contained in WINMEM32.DLL and explains how to use these functions in the context of a Windows application. It covers the following information:

- A brief look at some of the differences between a segmented memory model and a flat memory model
- Using WINMEM32.DLL to take advantage of the 32-bit capabilities of the 80386 and 80486 processors
- Programming considerations when using these capabilities in a Windows application
- Common approaches for using 32-bit memory in a Windows application

A directory of the functions supplied by WINMEM32.DLL follows this information. The appendix concludes with several assembly-language code examples illustrating how to use the DLL's functions.

IMPORTANT This appendix assumes that you are thoroughly familiar with the architecture and code- and memory-management features of the 80386/80486 processors. This appendix does not attempt to explain these features, and assumes that you are familiar with the terminology and concepts associated with that architecture.

Only experienced Windows-application developers with extensive experience writing assembly-level code should attempt to use these functions in an application.

E.1 Segmented and Flat Memory Models

The 80x86 family of microprocessors implement a segmented memory model in which system memory is divided into 64K segments. In the native mode of these processors, the address of any byte consists of two 16-bit values: a segment address and an offset. In the protected mode of the 80286, 80386, and 80486 processors, the segment address is replaced by a selector value which the processor uses to access the 64K segment. In both modes, memory objects larger than 64K will occupy all or part of several segments. An application cannot access these objects as though they consist of a single contiguous block simply by incrementing a pointer to the memory. Instead, the application can increment only the offset portion of the address, taking care not to exceed the 64K boundary of the segment.

The 80386 processor introduced 32-bit registers that parallel the 16-bit registers of older members of the 80x86 family. These registers make it possible for the first time to access memory in segments larger than 64K. In fact, the maximum segment size is potentially so large (2^{32} bytes) that a flat memory model utilizing a single segment is now feasible. In this model, an application's code and/or data occupies a single segment. The application can manipulate the 32-bit offset portion of the memory as though it were a simple pointer. The application can increment and decrement the pointer/offset throughout the address space without having to deal with multiple segment boundaries.

To a certain extent, then, the flat memory model most closely resembles the tiny memory model in which both code and data occupy a single segment; except, of course, that the segment is much larger than the 64K limit imposed by the segmented memory model. As in the tiny memory model, the beginning of the segment of the flat memory model can appear anywhere in memory. In other words, the segment-descriptor portion of the address can refer to virtually any location in memory. As the application moves through memory, the segment descriptor never changes. Only the offset is incremented and decremented to point to different locations in memory.

As this appendix will note, it is not possible to implement a Windows application using an exclusively flat memory model. Windows itself relies on the 16-bit segmented memory model, and so any application that interacts with Windows must implement at least one 16-bit code segment. Despite this limitation, however, it is possible for a Windows application to reside largely in one or more 32-bit code segments and to use 32-bit data segments. The WINMEM32.DLL library makes this possible in a way that ensures the application will cooperate fully with Windows and similar platforms.

E.2 Using the WINMEM32.DLL Library

Although you could directly implement flat memory model code in your Windows application, this implementation would necessarily be unique to your application. As a result, your application might not run with future versions of Windows or with other compatible platforms.

WINMEM32.DLL supplies a standard method for implementing a flat memory model that is guaranteed to run with future versions of Windows and other compatible platforms. It gives your application access to services for allocating, reallocating, and freeing 32-bit memory objects; for translating 32-bit pointers to 16-bit pointers that can be used by Windows and DOS functions; and for aliasing a data segment to a code segment so you can execute code loaded into a 32-bit segment.

Your application can load WINMEM32.DLL when Windows is running in real, standard, or 386 enhanced mode. However, since the 32-bit registers of the 80386/80486 processor are available only when Windows is in 386 enhanced mode, WINMEM32.DLL is enabled only in that mode. If your application can run in real or standard mode, you must design your application so that it can access 16-bit memory instead of 32-bit memory in these modes. You can determine the mode in which Windows is running by calling the **GetWinFlags** function.

WINMEM32.DLL contains eight functions that enable your application to access 32-bit memory. The following list summarizes each of these functions:

Function	Description
Global32Alloc	Allocates a block of 32-bit memory.
Global32Realloc	Changes the size of a 32-bit memory object.
Global32Free	Frees a 32-bit memory object.
Global16PointerAlloc	Converts a 32-bit pointer to a 16-bit pointer.
Global16PointerFree	Frees a pointer alias created by **Global16PointerAlloc.**
Global32CodeAlias	Creates a code alias for a 32-bit memory object, allowing code in the the object to be executed.
Global32CodeAliasFree	Frees a code alias created by **Global32CodeAlias.**
GetWinMem32Verwsion	Returns the version number of the WINMEM32.DLL API.

A directory listing of these functions appears later in this appendix.

WINMEM32.DLL is a standard Windows DLL, and so your application loads it as it would any other DLL. In addition to the DLL, the SDK provides the C-language WINMEM32.H include file to declare the functions in your application, and the import library WINMEM32.LIB to allow your application to import the functions of the DLL when you link your application.

The calling convention of the WINMEM32.DLL functions is the same as for other Windows functions. The DLL entry points are external **FAR PASCAL** procedures. They preserve SS, BP, DS, SI, and DI, and they return values in AX or DX:AX.

E.3 Considerations for Using 32-Bit Memory

As previously noted, Windows adheres to the segmented memory model. That is, all far pointers are in the form 16:16 consisting of a 16-bit segment address (in real mode) or selector (in protected mode), combined with a 16-bit offset within the segment. An application using the 32-bit registers of the 80386/80486 processor cannot directly call the Windows functions because its far pointers are in the form 16:32 and Windows cannot deal with the extra 16 bits in the offset portion of the address.

Because of this conflict, a Windows application cannot reside exclusively in a 32-bit segment. It must contain at least one 16-bit "helper" code segment through which it interacts with Windows (including WINMEM32.DLL). In other words, all calls to Windows functions must be made in the helper code segment. The helper segment contains the code that converts the 16:32 pointers in the 32-bit segment to the 16:16 pointers used by Windows functions. This segment also performs the same tasks for the application when the application is making calls to DOS, to other DLLs, and to any other code that exclusively uses 16:16 pointers.

An important limitation on this helper segment is that it must not be discardable. If the segment were discarded and a 32-bit segment were to attempt to access the segment, an indirect call into the Windows kernel module to reload the segment would result. Since the source of this indirect call would not be a 16-bit segment, the system might crash.

Another important consideration is that your application must not assume anything about the state of the 32-bit registers around 16:16 API calls. For instance, the Windows API calls preserve SI and DI, but they presently do not preserve ESI and EDI. If the application wants to preserve 32-bit registers around 16:16 API calls, it must explicitly push and pop the register values around the calls. If the 32-bit code segment that calls a Windows function (via the helper segment) assumes that ESI and EDI will be preserved when the Windows function returns, the helper segment must explicitly save the registers before making the actual Windows function call. The helper segment must then restore the registers when the Windows function returns.

This rule also applies to return values when a 32-bit segment indirectly calls a Windows function and the caller expects a 32-bit return value. The helper segment must explicitly set the high-order 16 bits of the return value when it moves it into the EAX register, as shown in the following examples:

```
MOVEZX    EAX,AX      ; Unsigned return

MOVESX    EAX,AX      ; Signed return
```

All these considerations apply equally to calls to Windows DLLs, DOS, and other 16-bit APIs.

E.3.1 The Flat Model Under Windows

In the Windows environment, system memory is a shared resource which Windows manages on behalf of all applications. For this reason, a true flat memory model is not possible in the Windows environment. When an application allocates 32-bit memory in Windows, the memory that Windows gives the application can be located anywhere in physical memory. The memory to which the selector refers is specific to the application and does not include system-wide memory locations. In other words, the selector that the application receives does not refer to interrupt vector 0. This means that offset 400h for the selector does not point to the DOS ROM BIOS data area, for example.

E.3.2 The Application Stack

Windows has problems operating in an environment of mixed segment types (both 16:16 and 16:32 segments). As a result, the stack selector size must match the corresponding code selector size. In other words, when the processor is executing code in a 16:32 (USE32) code segment, the selector in the SS register must also be 16:32. And when code in a 16:16 (USE 16) segment is executing, the SS register must contain a 16:16 selector.

When the 80386/80486 processor is executing on a USE16 stack segment, it uses the low-order 16 bits of the ESP register as the SP register. Since only the lower 16 bits are of use when the processor is executing on a USE16 stack segment, it does not control how the upper 16 bits of the ESP register are set. As a result, the upper 16 bits are set at random. When an application switches to a USE32 stack segment, the ESP register will contain a corrupted pointer unless the high 16 bits of ESP are set properly.

For example, a Windows application has a USE32 code segment and a USE16 helper segment, but (improperly) only a USE32 stack. When the application calls from its USE32 code into the USE16 segment, it stays on its USE32 stack. The USE16 code segment calls a Windows function, which changes the selector in the SS register to a USE16 selector. Since the stack is now USE16, the upper 16 bits of the ESP register are set at random. The code that originally switched stacks then restores the original selector in SS and, not knowing that it referred to a USE32 stack, restores the 16-bit SP register instead of the full 32 bits of the ESP register. As a result, the USE32 stack now has an invalid pointer in the ESP register.

There are a number of ways to deal with this problem. First, an application can maintain two separate stacks, one USE16 and the other USE32. Maintaining separate stacks requires you to include extra code —for example, you must copy parameters for stack-calling conventions such as C. Another solution would be to maintain one stack but two stack selectors, again one USE16 and the other USE32. Both selectors would point to the same USE32 memory. This would require the USE32 stack to be restricted to ESP values less than or equal to FFFFh.

In either case, the USE16 code segment must switch to the USE32 stack immediately before calling into a USE32 code segment. When control returns from the USE32 code segment to the USE16 code segment, the USE16 segment must immediately switch back to the USE16 stack before doing anything else.

Since the problem with stack switching is the corruption of the high 16 bits of ESP, a Windows application with 16:32 code must make sure that it sets the high 16 bits of ESP when it is switching onto the USE32 stack selector. It sets these bits by placing the selector into SS, as shown in the following example:

```
MOV     SS,word ptr [Use32StackSel]
MOV     ESP,dword ptr [Use32StackOffset]

MOV     SS,word ptr [Use32StackSel]
MOVZX   ESP,word ptr [Use32StackOffset]

MOV     SS,word ptr [Use32StackSel]
MOVZX   ESP,SP
```

E.3.3 Interrupt-Time Code

Because Windows is a 16-bit environment, Windows has problems dealing with a mixed code-type environment, a 32-bit code segment in a Windows application must not contain code that is executed at interrupt time. Also, it must not contain data that is accessed at interrupt time. Any code executed at interrupt time must be in a USE16 code segment running on a USE16 stack. Data used at interrupt time must be USE16 data. This rule also applies to processor exceptions (such as the coprocessor exception) since they are handled like interrupts. Note, however, that it is acceptable for a 32-bit code segment to access data in a USE16 data segment.

E.3.4 Programming Languages

As should be obvious by now, the helper segment has to perform very low-level tasks to manage transitions between USE16 and USE32 stacks, and between USE16 and USE32 code. For this reason, it is difficult to use a high-level language such as C to write the helper segment code. Even if you were to write the helper segment in C, you would have to add assembly-language support for the more difficult tasks. In most cases, then, it is easier and more efficient to write the entire helper segment in assembly language.

E.4 Using 32-Bit Memory in a Windows Application

There are three common uses for 32-bit memory in a Windows application. In increasing order of complexity, they are:

- Using 32-bit data objects in 16-bit code
- Using 32-bit code and data in a subroutine library
- Using 32-bit code and data for the main program

The following sections briefly describe each of these approaches.

E.4.1 Using 32-Bit Data Objects

The simplest use of 32-bit memory is to store data that is used exclusively by USE16 code segments. In this case, the application contains no USE32 code segments and so does not require a dedicated helper segment. Instead, any (or all) of its code segments performs the necessary tasks of allocating, reallocating, and freeing the 32-bit memory. If data from the 32-bit memory is to be passed to Windows functions or other 16-bit functions, the application's USE16 code segment also performs the aliasing of 32-bit pointers to 16-bit pointers using the **Global16PointerAlloc** function.

E.4.2 Using 32-Bit Code and Data in a Subroutine Library

Using 32-bit memory for code and data can simplify porting an application from a 32-bit platforms to the Windows environment when portions of the application can be isolated as a subroutine library. This subroutine library serves as a low-level engine, but does not call Windows or DOS functions.

As when the 32-bit memory is used exclusively for data storage, the USE16 code segment retains control of the program. Typically, the USE16 segment allocates the 32-bit memory, creating one or more objects for code and data. In addition to the data-management tasks described in the previous section, the USE16 segment also loads the subroutine code into one of the 32-bit segments, fixes up the pointers in the code as required, and creates a code-segment alias to permit the code to be executed. The USE16 code segment is responsible for maintaining control of the program flow, calling into the USE32 code segment when it requires the low-level services of the subroutine library.

E.4.3 Using 32-Bit Code and Data for the Main Program

The most complex use of 32-bit memory involves placing the primary control of the program in a 32-bit code segment. In this type of application, the USE16 segment is reduced exclusively to helper status. During initialization, the USE16 segment allocates the 32-bit memory for code and data, loads the code into the USE32 segment, creates a code-segment alias for the USE32 segment, and then calls the main entry point in the USE32 segment.

From that point, the USE32 segment takes control of the program, calling into the USE16 helper segment only when the application needs to call Windows or DOS functions. The USE32 segment continues to control the flow of the program until the application is ready to terminate. Only then does it return control to the USE16 segment so the USE16 segment can free the 32-bit memory and perform other garbage collection before the application quits.

E.5 Functions Directory

This section describes the functions in WINMEM32.DLL. Most of these functions return zero to indicate success or a nonzero error-code value to indicate failure. The following list describes these error codes.

Value	Meaning
1	Invalid function. The current Windows mode does not support this function. Windows supports the 32-bit memory functions only in 386 enhanced mode.
2	Invalid flags. The *wFlags* parameter contained invalid bit settings. The *wFlags* parameter currently is not used and must be set to zero.
3	Invalid parameter. One of the parameters was invalid. For example, a size parameter was out of range.
4	Selector not available. There is not enough room in the descriptor table(s) to allocate the required selector(s). It may be necessary to advise the user to close other Windows applications.
5	Insufficient memory. There is not enough memory to satisfy the requested allocation or reallocation.

GetWinMem32Version 3.0

Syntax **WORD GetWinMem32Version( )**

This function returns the API version implemented by the DLL. This is not the version number of the DLL itself.

This function has no parameters.

Return Value The return value specifies the version of the 32-bit memory API implemented by WINMEM32.DLL. The high-order 8 bits contain the major version number, and the low-order 8 bits contain the minor version number. The current API version number is 1.00 (100h): the major version number is 1, and the minor version number is 0.

Global16PointerAlloc 3.0

Syntax **WORD Global16PointerAlloc**(*wSelector*, *dwOffset*, *lpBuffer*, *dwSize*, *wFlags*)

This function converts a 16:32 pointer into a 16:16 pointer alias that the application can pass to a Windows function or other 16:16 functions.

Parameter	Type/Description
wSelector	**WORD** Specifies the selector of the object for which an alias is to be created. This must be the selector returned by a previous call to **Global32Alloc**.
dwOffset	**DWORD** Specifies the offset of the first byte for which an alias is to be created. The offset is from the first byte of the object specified by the *wSelector* parameter. Note that *wSelector:dwOffset* forms a 16:32 address of the first byte of the region for which an alias is to be created.
lpBuffer	**LPDWORD** Points to a four-byte location in memory that receives the 16:16 pointer alias for the specified region.
dwSize	**DWORD** Specifies the addressable size in bytes of the region for which an alias is to be created. This value must be in the range 1 to 10000h.
wFlags	**WORD** Is reserved and must be set to zero.

Return Value The return value is zero if the function was successful. Otherwise, it is one of the error codes described at the beginning of this section.

Comments

When this function returns successfully, the location pointed to by the *lpBuffer* parameter contains a 16:16 pointer to the first byte of the region. This is the same byte to which *wSelector:dwOffset* points.

The returned selector is a read/write, expand up, small (B bit clear) data descriptor. The descriptor DPL and the setting of granularity (the G bit) are at the discretion of the system, and so the application should not assume their settings. The descriptor DPL and the selector RPL are appropriate for a Windows application.

NOTE An application must not change the setting of any fields in the descriptor or the selector RPL. Doing so can result in a system crash and will prevent the application from running on compatible platforms.

Because of tiling schemes implemented by some systems, the offset portion of the returned 16:16 pointer is not necessarily zero.

An application should not assume the size limit of the returned selector. Instead, an application should assume that at least *dwSize* bytes can be addressed starting at the 16:16 pointer created by this function.

Global16PointerFree 3.0

Syntax

WORD Global16PointerFree(*wSelector*, *dwAlias*, *wFlags*)

This function frees the 16:16 pointer alias previously created by a call to the **Global16PointerAlloc** function.

Parameter	Type/Description
wSelector	**WORD** Specifies the selector of the object for which the alias is to be freed. This must be the selector returned by a previous call to **Global32Alloc**.
dwAlias	**DWORD** Specifies the 16:16 pointer alias to be freed. This must be the alias (including the original offset) returned by a previous call to **Global16PointerAlloc**.
wFlags	**WORD** Is reserved and must be set to zero.

Return Value

The return value is zero if the function was successful. Otherwise, it is one of the error codes described at the beginning of this section.

Comments

An application should free a 16:16 pointer alias as soon as it is no longer needed. Freeing the alias releases space in the descriptor table, a limited system resource.

Global32Alloc 3.0

Syntax **WORD Global32Alloc**(*dwSize*, *lpSelector*, *dwMaxSize*, *wFlags*)

This function allocates a block of memory to be used as a USE32 code or data segment and retrieves the selector portion of the 16:32 address of the memory block. The first byte of the object is at offset 0 from this selector.

Parameter	Type/Description
dwSize	**DWORD** Specifies the initial size in bytes of the block to be allocated. This value must be in the range of 1 to 4000000h (64 megabytes).
lpSelector	**LPWORD** Points to a two-byte location in memory that receives the selector portion of the 16:32 address of the allocated object.
dwMaxSize	**DWORD** Specifies the maximum size in bytes that the object will reach when it is reallocated by the **Global32Realloc** function. This value must be in the range of 1 to 4000000h (64 megabytes). If the application will never reallocate this block of memory, the *dwMaxSize* parameter should be set to the same value as the *dwSize* parameter.
wFlags	**WORD** Is reserved and must be set to zero.

Return Value The return value is zero if the function was successful. Otherwise, it is one of the error codes described at the beginning of this section.

Comments If the **Global32Alloc** function fails, the value to which *lpSelector* points is zero. If the function succeeds, *lpSelector* points to the selector of the object. The valid range of offsets for the object referenced by this selector is in the range of 0 to (but not including) *dwSize*.

The returned selector is a read/write, expand-up, big (B bit set), data descriptor. The descriptor DPL and the setting of granularity (the G bit) are at the discretion of the system; the application should not assume these settings. Since the system sets the granularity, the actual size of the object (and the selector size limit) may be greater than the requested size by up to one byte less than 4K. The descriptor DPL and the selector RPL will be appropriate for a Windows application.

NOTE An application must not change the setting of any fields in the descriptor or the selector RPL. Doing so can result in a system crash and will prevent the application from running on compatible platforms.

The allocated object is neither moveable nor discardable, but can be paged. Since page locking an object is useful only if the object contains code or data that is used at interrupt time, and since 32-bit memory cannot be used at interrupt time, an application should not page lock a 32-bit memory object.

Global32CodeAlias 3.0

Syntax

WORD **Global32CodeAlias**(*wSelector*, *lpAlias*, *wFlags*)

This function creates a 16:32 (USE32) code alias selector for a 32-bit memory object previously created by the **Global32Alloc** function. This allows the application to execute code contained in the memory object.

Parameter	Type/Description
wSelector	**WORD** Specifies the selector of the object for which an alias is to be created. This must be the selector returned by a previous call to **Global32Alloc.**
lpAlias	**LPWORD** Points to a two-byte location in memory that receives the 16:32 code-segment alias selector for the specified object.
wFlags	**WORD** Is reserved and must be set to zero.

Return Value

The return value is zero if the function was successful. Otherwise, it is one of the error codes described at the beginning of this section.

Comments

If the function fails, the value pointed to by the *lpAlias* parameter is zero. If the function is successful, *lpAlias* points to a USE32 code-segment alias for the object specified by the *wSelector* parameter. The first byte of the object is at offset 0 from the selector returned in *lpAlias*. Valid offsets are determined by the size of the object as set by the most recent call to the **Global32Alloc** or **Global32Realloc** function.

The returned selector is a read/execute, nonconforming, USE32 (D bit set) code descriptor. The descriptor DPL and the setting of granularity (the G bit) are at the discretion of the system, and so the application should not assume their settings. The granularity will be consistent with the current data selector for the object. The descriptor DPL and the selector RPL are appropriate for a Windows application.

NOTE An application must not change the setting of any fields in the descriptor or the selector RPL. Doing so can result in a system crash and will prevent the application from running on compatible platforms.

An application should not call this function more than once for an object. Depending on the system, the function might fail if an application calls it a second time for a given object without first calling the **Global32CodeAliasFree** function for the object.

Global32CodeAliasFree 3.0

Syntax WORD **Global32CodeAliasFree**(*wSelector*, *wAlias*, *wFlags*)

This function frees the USE32 code selector alias previously created by a call to the **Global32CodeAlias** function.

Parameter	Type/Description
wSelector	**WORD** Specifies the selector of the object for which the alias is to be freed. This must be the selector returned by a previous call to **Global32Alloc.**
wAlias	**WORD** Specifies the USE32 code selector alias to be freed. This must be the alias returned by a previous call to **Global32CodeAlias.**
wFlags	**WORD** Is reserved and must be set to zero.

Return Value The return value is zero if the function was successful. Otherwise, it is one of the error codes described at the beginning of this section.

Global32Free 3.0

Syntax WORD **Global32Free**(*wSelector*, *wFlags*)

This function frees an object previously allocated by the **Global32Alloc** function.

Parameter	Type/Description
wSelector	**WORD** Specifies the selector of the object to be freed. This must be the selector returned by a previous call to **Global32Alloc.**
wFlags	**WORD** Is reserved and must be set to zero.

Return Value The return value is zero if the function was successful. Otherwise, it is one of the error codes described at the beginning of this section.

Comments This function frees the object itself, as well as all aliases created for the object by the 32-bit memory API.

NOTE Before terminating, an application must call this function to free each object allocated by **Global32Alloc** to ensure that all aliases created for the object are freed.

Global32Realloc 3.0

Syntax WORD **Global32Realloc**(*wSelector*, *dwNewSize*, *wFlags*)

This function changes the size of a 32-bit memory object previously allocated by the **Global32Alloc** function.

Parameter	Type/Description
wSelector	WORD Specifies the selector of the object to be changed. This must be the selector returned by a previous call to **Global32Alloc**.
dwNewSize	DWORD Specifies the new size in bytes of the object. This value must be greater than zero and less than or equal to the size specified by the *dwMaxSize* parameter of the **Global32Alloc** function call that created the object.
wFlags	WORD Is reserved and must be set to zero.

Return Value The return value is zero if the function was successful. Otherwise, it is one of the error codes described at the beginning of this section.

Comments If this function fails, the previous state of the object is unchanged. If the function succeeds, it updates the state of the object and the state of all aliases to the object created by the 32-bit memory API. For this reason, an application must call the **Global32Realloc** to change the size of the object. Using other Windows functions to manipulate the object will result in corrupted aliases.

This function does not change the selector specified by the *wSelector* parameter. If this function succeeds, the new valid range of offsets for the selector is in the range of 0 to (but not including) *dwNewSize*.

The system determines the appropriate granularity of the object. As a result, the actual size of the object (and the selector size limit) may be greater than the requested size by up to one byte less than 4K.

```
;------------------------------------------------------------------------------
;
;   SAMPLE code for WINMEM32 DLL
;
;------------------------------------------------------------------------------

.386p

memS    equ     1

        .xlist

        include    cmacros.inc
;
; NOTE that we CANNOT use the normal CMACROS segment macros:
;
;       CreateSeg
;       sBegin
;       sEnd
;
; because since we are .386p the default segment type is USE32. Our segments
; need to be USE16 so we have to declare our segements manually so that the
; USE16 segment attribute is included.
;
        include    windows.inc

        .list
;
; These equates would normally be in an app specific include file
;
error_bad_file    EQU   08001h
error_wrong_mode  EQU   08002h


;------------------------------------------------------------------------------
;
; External WINMEM32 Procedures
;
externFP        GetWinMem32Version
externFP        Global32Alloc
externFP        Global32Realloc
externFP        Global32Free
externFP        Global16PointerAlloc
externFP        Global16PointerFree
externFP        Global32CodeAlias
externFP        Global32CodeAliasFree

;------------------------------------------------------------------------------
;
; External Windows Procedures
;
externFP        OpenFile
externFP        GetWinFlags
```

```
externFP        _llseek
externFP        _lread
externFP        _lclose
externFP        OemToAnsi


;
; MANUAL VERSION OF: createSeg _HELPERCODE,hcode,word,public,CODE
;
; NOTE that this segment MUST NOT BE DISCARDABLE, it should be fixed.
;      This is because the segment is called by USE32 code.
;
_HELPERCODE segment word public 'CODE' use16
_HELPERCODE ends


;
; MANUAL VERSION of the automatic data segment declaration
;
_DATA segment word public 'DATA' use16
_DATA ends

_DATA segment use16

globalD         AddrOEMToANSI,0         ; Address of OEMToANSI helper function
globalD         AddrDOSGetFreeSpace,0   ; Address of DOS Get disk Free space
                                        ;    helper function
globalD         U32RetVal,0             ; Return code from USE32 call

globalD         U16StackAlias,0         ; Alias for the stack

globalD         EntryStackSave,0        ; stack ptr save location

;
; This FWORD forms the entry point for the USE32 code
;
U32EntryPt      LABEL FWORD
globalD         U32EntOff,00010000h     ; Entry assumed at offset 64K
globalW         U32CodeSel,0            ; CODE alias for the BIG object

globalW         U32DataSel,0            ; DATA selector for the BIG object

_DATA ends

_HELPERCODE segment use16
        assume  cs:_HELPERCODE

;****************************************************************************
;
; SetupCallUSE32
;
;       SetupCallUSE32(fName)
;
;       Setup and call into USE32 code
```

```
;
; ASSUMPTIONS:
;       The USE32 Image is a 0 ORGed 32 bit code image with NO HEADER.
;       The first 64k of the image (offsets 00000000-0000FFFFh) is reserved
;         for the stack. We put the stack here so that the required stack
;         switching (USE32<->USE16) is simply a matter of changing SS.
;
;       The entry point of the USE32 code is assumed to be right after the
;         stack at offset 00010000h in the image. We enter with DS, FS, GS
;         and SS set to the FLAT data segment, and CS set to the flat code
;         segment. It is the responsibility of the USE32 entry point to set
;         ES AND PRESERVE IT FOR US.
;
;       When this code wishes to call the two provided USE16 helper routines,
;         it looks up the call addresses in the AddrOEMToANSI and
;         AddrDOSGetFreeSpace variables in the _DATA segment.
;         This "loader" code actually needs to pass the selector for the
;         _DATA segment to the USE32 code so that it can access the data
;         segment, or it needs to copy the call addresses into the USE32
;         code/data segment. This detail of the implementation is NOT
;         included in this code.
;
; ENTRY:
;       FName - DWORD pointer to file name of USE32 image to load
;
; EXIT:
;       AX != 0 If an error occurs
;           AX = error code
;       Else
;           AX = 0 and U32RetVal contains the return code from the
;                   USE32 code.
; USES:
;       C Standard
;
;*****************************************************************************
cProc   StartupCallUSE32,<FAR,PUBLIC>,<si,di>

        ParmD   fName

        LocalD  fSize                   ; Size of file
        LocalD  U16RdAlias              ; Alias for reading image
        LocalD  FileOff                 ; Current file offset for read
        LocalW  fHand                   ; File handle
        LocalV  OpnBuf,<SIZE OPENSTRUC> ; Open file struct for openfile call

cBegin
    assume  ds:_DATA
    assume  es:nothing
    assume  ss:_DATA
    ;
    ; First check if we are running in enhanced mode
    ;
    ; NOTE THAT SINCE WE DO NOT KNOW AS YET WHAT MODE WE ARE IN WE NEED
```

```
        ;   TO AVOID USING 386 SPECIFIC INSTRUCTIONS
        ;
            cCall   GetWinFlags

            and     ax,WF_PMODE + WF_ENHANCED
            cmp     ax,WF_PMODE + WF_ENHANCED
            je      short OKtoLoad                  ; MUST BE SHORT
            mov     ax,error_wrong_mode
            jmp     Done1

        ;
        ; We now know we are in the proper mode and that 386 instructions
        ;   are now OK.
        ;
OKtoLoad:
        ;
        ; Set up the addresses for the USE32 code to call the helper routines
        ;
            mov     ax,cs
            mov     word ptr [AddrOEMToANSI+2],ax
            mov     word ptr [AddrOEMToANSI],offset _HELPERCODE:U32OEMtoANSI
            mov     word ptr [AddrDOSGetFreeSpace+2],ax
            mov     word ptr [AddrDOSGetFreeSpace],offset _HELPERCODE:U32GetDskFree
        ;
        ; Open the file
        ;
            lea     bx,OpnBuf
            regptr  ssbx,ss,bx
            cCall   OpenFile,<fName,ssbx,OF_READ>
            cmp     ax,-1                       ; Did we find it?
            je      Done1FlErr                  ; No, file error
            mov     fHand,ax                    ; Save file handle
        ;
        ; Get file size
        ;
            cCall   _llseek,<fHand,0,0,2>
            shl     edx,16
            mov     dx,ax
            inc     edx
            jz      Done1FlErr                  ; seek failed, file error
            dec     edx
            mov     fSize,edx
            cmp     edx,10000h                  ; Image is at least 64k?
            jbe     Done1FlErr                  ; No, size is too small, file error
        ;
        ; Move file pointer back to start of file for read
        ;
            cCall   _llseek,<fHand,0,0,0>
        ;
        ; Allocate big USE32 object
        ;
            mov     si,dataOffset U32DataSel
            regptr  Selpt,ds,si
```

```
        cCall   Global32Alloc,<fSize,Selpt,fSize,Ø>
        or      ax,ax                   ; Worked?
        jnz     FcloseEr                ; No, return WINMEM32 error code
    ;
    ; Allocate USE16 stack alias for first 64K of object
    ;
        mov     si,dataOffset U16StackAlias
        regptr  Alipt,ds,si
        mov     ecx,ØØØ1ØØØØh           ; 64K
        cCall   Global16PointerAlloc,<[U32DataSel],Ø,Ø,Alipt,ecx,Ø>
        or      ax,ax                   ; Worked?
        jnz     AliasErrF3              ; No, return WINMEM32 error code
    ;
    ; Allocate USE32 code alias
    ;
        mov     si,dataOffset U32CodeSel
        regptr  Alipt,ds,si
        cCall   Global32CodeAlias,<[U32DataSel],Alipt,Ø>
        or      ax,ax                   ; Worked?
        jnz     AliasErrF2              ; No, return WINMEM32 error code
    ;
    ; Now read in the image. We will do this in 32K hunks.
    ;
        mov     FileOff,Ø               ; Starting at file offset Ø
ReadLp:
        mov     ecx,ØØØØ8ØØØh           ; 32k
        cmp     ecx,fSize
        jbe     short Read32k
        mov     ecx,fSize
Read32k:
    ;
    ; Make a USE16 alias for this region of the object
    ;
        push    ecx
        lea     si,U16RdAlias
        regptr  Alipt,ss,si
        cCall   Global16PointerAlloc,<[U32DataSel],FileOff,Alipt,ecx,Ø>
        pop     ecx
        or      ax,ax
        jnz     short AliasErrF1
        push    ecx
        cCall   _lread,<fHand,U16RdAlias,cx>
        push    ax
        cCall   Global16PointerFree,<[U32DataSel],U16RdAlias,Ø>
        pop     ax
        pop     ecx
        inc     ax
        jz      short FlRdErr
        dec     ax
        cmp     ax,cx
        jne     short FlRdErr
        add     FileOff,ecx
```

```
        sub     fSize,ecx
        ja      short ReadLp

    ;
    ; We are now ready to set up and call into the USE32 code
    ;
    ;   Save the current stack so we can switch to the USE32 stack
    ;
    ;   NOTE CAREFULLY THAT THIS MAKES THIS ROUTINE NON-REENTRANT
    ;   SINCE IT SAVES THE CURRENT SS:SP IN A STATIC MEMORY LOCATION.
    ;
        mov     word ptr [EntryStackSave],sp
        mov     word ptr [EntryStackSave+2],ss
        mov     ax,[U32DataSel]
        push    ds
        pop     es
    assume es:_DATA
    ;
    ; Set up all the segs, and call into USE32
    ;
    ; NOTE that we just leave the file open across the call.
    ;
        mov     ds,ax
    assume ds:nothing
        mov     fs,ax
        mov     gs,ax
        mov     ss,ax
    assume ss:nothing
        mov     esp,0000FFFCh
        call    [U32EntryPt]
    ;
    ; Recover DS and stack.
    ;
        mov     bx,es
        mov     ds,bx
    assume ds:_DATA
        mov     ss,word ptr [EntryStackSave+2]
    assume ss:_DATA
        mov     sp,word ptr [EntryStackSave]
    ;
    ; Set success return and clean up.
    ;
        mov     [U32RetVal],eax
        xor     ax,ax                   ; Return success
        jmp     short AliasErrFl

FlRdErr:
        mov     ax,error_bad_file
AliasErrFl:
    ;
    ; Free USE32 code alias
    ;
        push    ax                      ; Save error code
```

```
        cCall   Global32CodeAliasFree,<[U32DataSel],[U32CodeSel],0>
        pop     ax
AliasErrF2:
    ;
    ; Free USE16 stack alias
    ;
        push    ax                      ; Save error code
        cCall   Global16PointerFree,<[U32DataSel],[U16StackAlias],0>
        pop     ax
AliasErrF3:
    ;
    ; Free the object
    ;
        push    ax                      ; Save error code
        cCall   Global32Free,<[U32DataSel],0>
        pop     ax
FcloseEr:
    ;
    ; Close the file
    ;
        push    ax                      ; Save error code
        cCall   _lclose,<fHand>
        pop     ax
        jmp     short Done1

Done1FlErr:
        mov     ax,error_bad_file
Done1:
cEnd


;*****************************************************************************
;
; U32OEMtoANSI - Call OemToANSI from USE32 segment
;
;    Assumes PASCAL calling convention
;
; ENTRY:
;       U32OEMtoANSI(lpOemStr,lpAnsiStr)
;
;       NOTE that these pointer arguments are NOT really LPSTRs. They
;       are near pointers into the USE32 data object (implied segment
;       is U32DataSel)
;
; EXIT:
;       EAX is return code
;
; USES:
;       32 bit C Standard
;
;
;*****************************************************************************
PUBLIC U32OEMtoANSI
```

```
U32OEMtoANSI proc far
    assume  ds:nothing
    assume  es:nothing
    assume  ss:nothing
    ;
    ; First switch to the USE16 stack
    ;
        mov     cx,ds           ; Save entry DS in cx till we get the stack switched
        mov     ax,SEG _DATA
        mov     ds,ax
    assume ds:_DATA
        mov     ss,word ptr [U16StackAlias+2]
        push    ecx             ; Entry DS, as DWORD to keep stack aligned
        push    ebp
        mov     bp,sp
    ;
    ; Frame now looks like this:
    ;
    ;   dword ptr [bp + 20]  -->  First arg to OEMToANSI  lpOemStr (actually a 32
bit near pointer)
    ;   dword ptr [bp + 16]  -->  Second arg to OEMToANSI lpAnsiStr (actually a 32
bit near pointer)
    ;   dword ptr [bp + 12]  -->  Return CS
    ;   dword ptr [bp + 8]   -->  Return EIP
    ;   dword ptr [bp + 4]   -->  Entry DS pushed as DWORD
    ;   dword ptr [bp + 0]   -->  Entry EBP
    ;
lpOemStr        equ     dword ptr [bp+20]
lpAnsiStr       equ     dword ptr [bp+16]

        sub     sp,8            ; Need two LPSTRs for the aliases

AlsOemStr       equ     dword ptr [bp-4]        ; Alias for lpOemStr
AlsAnsiStr      equ     dword ptr [bp-8]        ; Alias for lpAnsiStr

        push    esi
        push    edi
        push    ebx
        push    es              ; Preserve "flat" ES, FS, GS
        push    fs
        push    gs
    ;
    ; There is a ?, how BIG is lpOemStr? Need to know this to set the
    ;   size of the alias(s). What we will do is "cheat". We will set
    ;   the size to 64k (or size to end of USE32 object, whichever is
    ;   smaller). NOTE that this assumes that the string is <= 64K which
    ;   is a reasonable assumption since we can't alias something larger
    ;   than that anyway.
    ;
        lsl     eax,dword ptr [U32DataSel]  ; Get limit of USE32 object
        inc     eax             ; Limit -> size
        mov     edx,eax
```

```
        sub     eax,lpOemStr     ; Number of bytes to end of USE32 object
        jc      SkipCall         ; Bad string ptr
        sub     edx,lpAnsiStr    ; Number of bytes to end of USE32 object
        jc      short SkipCall   ; Bad string ptr
        cmp     eax,edx
        jbe     short UseSrcLim
        mov     eax,edx          ; lpAnsiStr is closer to end of object
UseSrcLim:
        mov     ecx,00010000h    ; 64k
        cmp     ecx,eax
        jbe     short Use64k
        mov     ecx,eax          ; Limited by size to end of object
Use64k:
    ;
    ; Create Alias for lpOemStr
    ;
        push    ecx
        lea     bx,AlsOemStr
        regptr  AlsPt,ss,bx

        cCall   Global16PointerAlloc,<[U32DataSel],lpOemStr,AlsPt,ecx,0>

        pop     ecx
        or      ax,ax
        jnz     short SkipCall
    ;
    ; Create Alias for lpAnsiStr
    ;
        lea     bx,AlsAnsiStr

        cCall   Global16PointerAlloc,<[U32DataSel],lpAnsiStr,AlsPt,ecx,0>

        or      ax,ax
        jnz     short FreeOemAls
    ;
    ; Call OemToAnsi
    ;
        cCall   OemToAnsi,<AlsOemStr,AlsAnsiStr>
    ;
    ; Free the aliases
    ;
        push    ax               ; Save RET code

        cCall   Global16PointerFree,<[U32DataSel],AlsAnsiStr,0>

        pop     ax               ; Restore RET code
FreeOemAls:
        push    ax               ; Save RET code

        cCall   Global16PointerFree,<[U32DataSel],AlsOemStr,0>

        pop     ax               ; Restore RET code
SkipCall:
```

```
        pop     gs
        pop     fs
        pop     es
        pop     ebx
        pop     edi
        pop     esi
        add     sp,8
        pop     ebp
        pop     ecx                 ; Entry DS in CX
    ;
    ; Sign extend the return to make it 32 bit
    ;
        movsx   eax,ax
    ;
    ; Switch back to the USE32 stack MAKING SURE TO SET HIGH 16 BITS OF ESP.
    ;
        mov     ss,[U32DataSel]
        movzx   esp,sp
        mov     ds,cx
    assume  ds:nothing
        db      66h                 ; USE32 override on far ret so it returns to EIP
        ret     (2 * 4)

U32OEMtoANSI endp


;****************************************************************************
;
; U32GetDskFree - Issue DOS call to get disk free space
;
;    Assumes PASCAL calling convention
;
; ENTRY:
;       U32GetDskFree(drvnum)
;
; EXIT:
;       EAX = Disk free space in bytes
;       EAX == ØFFFFFFFFh if error
;
; USES:
;       32 bit C Standard
;
;
;****************************************************************************
PUBLIC U32GetDskFree

U32GetDskFree proc far
    assume  ds:nothing
    assume  es:nothing
    assume  ss:nothing
    ;
    ; First switch to the USE16 stack
    ;
```

```
        mov     cx,ds               ; Save entry DS in cx till we get the stack switched
        mov     ax,SEG _DATA
        mov     ds,ax
    assume ds:_DATA
        mov     ss,word ptr [U16StackAlias+2]
        push    ecx                 ; Entry DS, as DWORD to keep stack aligned
        push    ebp
        mov     bp,sp
    ;
    ; Frame now looks like this:
    ;
    ;   dword ptr [bp + 16]  -->  Drive # argument (Ø = default, A = 1 ...)
    ;   dword ptr [bp + 12]  -->  Return CS
    ;   dword ptr [bp + 8]   -->  Return EIP
    ;   dword ptr [bp + 4]   -->  Entry DS pushed as DWORD
    ;   dword ptr [bp + Ø]   -->  Entry EBP
    ;
ArgDrv          equ     dword ptr [bp+16]

        push    esi
        push    edi
        push    ebx
        push    es                  ; Preserve "flat" ES, FS, GS
        push    fs
        push    gs

        mov     edx,ArgDrv          ; Drive # to DL
        mov     ah,36h
        int     21h                 ; Make DOS call

        movsx   eax,ax              ; Sign extend AX for error
        cmp     ax,ØFFFFh           ; Error?
        je      short BadDrv        ; Yes, return ØFFFFFFFFh
        movzx   eax,ax              ; Convert sectors/cluster to 32 bit
        movzx   ebx,bx              ; Convert Available clusters to 32 bit
        movzx   ecx,cx              ; Convert bytes/sector to 32 bit
        mul     ecx                 ; EAX = sectors/cluster * bytes/sector =
                                    ;       bytes/cluster
        mul     ebx                 ; EAX = bytes/cluster * Available clusters =
                                    ;       free bytes
BadDrv:
        pop     gs
        pop     fs
        pop     es
        pop     ebx
        pop     edi
        pop     esi
        pop     ebp
        pop     ecx                 ; Entry DS in CX
    ;
    ; Switch back to the USE32 stack MAKING SURE TO SET HIGH 16 BITS OF ESP.
    ;
        mov     ss,[U32DataSel]
```

```
        movzx   esp,sp
        mov     ds,cx
    assume  ds:nothing
        db      66h                 ; USE32 override on far ret so it returns to EIP
        ret     (1 * 4)

U32GetDskFree endp

_HELPERCODE ends

        end
```

Index

Special Characters

A

B

C

D

E

F

H

I

K

L

M

N

O

P

S

T

U

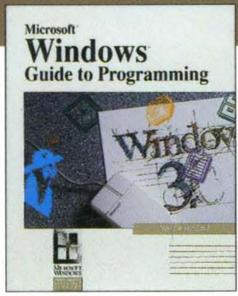
V

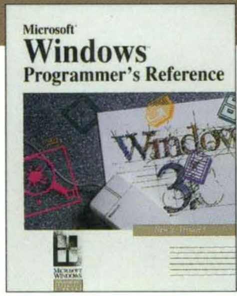
W

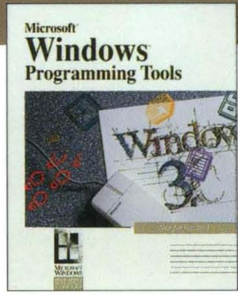
Y

Microsoft

Look for all three books in the Microsoft Windows™ Programmer's Reference Library.







Here, for every Windows programmer, is an up-to-date, comprehensive reference to each component in the Windows application programming interface (API). Included is detailed information on each Windows environment version 3 function, message, data type, resource-compiler statement, assembly-language macro, and file format.

This information is the foundation for any program that takes advantage of special Windows capabilities: data interchange with other applications; device-independent graphics; multitasking; dynamic linking; and shared display, memory, keyboard, mouse, and system timer resources.

Look for these other books in the Microsoft Windows Programmer's Reference Library:

Microsoft Windows Guide to Programming. An example-packed introduction to writing applications using version 3 of the Microsoft Windows application programming interface (API). Step-by-step instructions accompanied by dozens of code excerpts.

Microsoft Windows Programming Tools. Detailed instructions on using the specialized Windows software development tools: the C Compiler, the linker, the Resource Compiler, the resource editors, and the debugging and optimization tools. A special section addresses the Windows Help system and its programming guidelines.

Please Note: *The three books in the Microsoft Windows Programmer's Reference Library are included in the Microsoft Windows Software Development Kit (SDK).*

U.S.A.	**$39.95**
U.K.	£36.95
Canada	$44.95

[*Recommended*]

Microsoft PRESS®

The Authorized Editions

ISBN 1-55615-309-0

90000

9 781556 153099

Programming/Windows/C