PROGRAMMER'S REFERENCE LIBRARY

MICROSOFT® WINDOWS™

Microsoft® Windows™ 3.1 Programmer's Reference

Volume 1

Overview

Microsoft® PRESS

Microsoft® Windows™ 3.1

Programmer's Reference

Volume 1

Overview

Microsoft PRESS®

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

Library of Congress Cataloging-in-Publication Data
Microsoft Windows programmer's reference.
p. cm.
Includes indexes.
Contents: v. 1. Overview -- v. 2. Functions -- v. 3. Messages, structures, macros -- v. 4. Resources.
ISBN 1-55615-453-4 (v. 1). -- ISBN 1-55615-463-1 (v. 2). -- ISBN 1-55615-464-X (v. 3). -- ISBN 1-55615-494-1 (v. 4)
1. Microsoft Windows (Computer program) I. Microsoft Corporation.
QA76.76.W56M532 1992
005.4'3--dc20 91-34199
CIP

Printed and bound in the United States of America.

1 2 3 4 5 6 7 8 9 MLML 7 6 5 4 3 2

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

British Cataloging-in-Publication Data available.

Document No. PC28915-0492

Contents

Introduction

This manual, *Microsoft Windows Programmer's Reference*, *Volume 1*, describes different interface functions and extension libraries supported by the Microsoft® Windows™ operating system. It also includes application notes describing special Windows features for applications. The appendix provides a listing of module and library names for Windows functions.

Part 1, "Windows Management, Graphics, and Systems Services," presents functions that relate to window management, graphics output, and system services. Window manager functions process messages; create, move, or alter a window; or create system output. Graphics device interface (GDI) functions perform device-independent graphics operations, such as the creation of line, text, and bitmap output on different output devices. System services functions perform operations such as accessing code and data in modules, allocating and managing memory, translating strings, and creating and opening files.

Part 2, "Extension Libraries," describes the libraries that support many of the features new to Windows version 3.1. These new features include common dialog boxes; management functions that simplify dynamic data exchange (DDE); object linking and embedding (OLE); such shell enhancements as the registration database and the drag-drop feature; tool helper functions that streamline the creation of Windows-hosted tools; data decompression functions; a stress-testing facility that artificially consumes system resources and can be used when debugging applications; file installation functions; functions that allow an application to make use of the 32-bit memory-addressing capabilities of 80386 and 80486 processors; floating-point emulation; and the screen saver that is built into Control Panel.

Part 3, "Application Notes," describes techniques an application should use to implement some Windows features and enhancements. This part of the manual explains how to create a Control Panel application, how to create and install extensions for File Manager, how to use the dynamic-data exchange interface of Program Manager, how to make applications country- and language-independent, how to write network applications, how to integrate Windows applications with Microsoft MS-DOS® functions, how to write a compiler that generates Windows prolog and epilog code, how to initialize and start Windows applications, how to improve the video performance of Windows applications, how to write self-loading Windows applications, and how to interact with installable drivers.

The appendix lists the module and library for each Windows function.

Document Conventions

The following conventions are used throughout this manual to define syntax:

Convention	Meaning
Bold text	Denotes a term or character to be typed literally, such as a resource-definition statement or function name (**MENU** or **CreateWindow**), an MS-DOS command, or a command-line option (**/nod**). You must type these terms exactly as shown.
Italic text	Denotes a placeholder or variable: You must provide the actual value. For example, the statement **SetCursorPos**(*X*, *Y*) requires you to substitute values for the *X* and *Y* parameters.
[ ]	Enclose optional parameters.
|	Separates an either/or choice.
...	Specifies that the preceding item may be repeated.
`BEGIN` . . . `END`	Represents an omitted portion of a sample application.

In addition, certain text conventions are used to help you understand this material:

Convention	Meaning
SMALL CAPITALS	Indicate the names of keys, key sequences, and key combinations—for example, ALT+SPACEBAR.
FULL CAPITALS	Indicate filenames and paths, most type and structure names (which are also bold), and constants.
`monospace`	Sets off code examples and shows syntax spacing.

Window Management, Graphics, and System Services

Part 1

Window Management

Chapter 1

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

1.1 Messages

Messages are the input to an application. They represent events that the application may need to respond to. A message is a structure that contains a message identifier and message parameters. The content of the parameters varies with the message type.

1.1.1 Generating and Processing Messages

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

Windows sends some messages directly to the window procedure in the appropriate application instead of placing the messages in the application's message queue. Such messages are called unqueued messages. Typically, an unqueued message is any message that affects the window only. The **SendMessage** function sends messages directly to a window procedure. For more information about window procedures, see Section 1.2.13, "Window Procedures."

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

Although Windows generates most messages, an application can create its own messages and place them in its own message queue or that of another application.

An application typically uses the **GetMessage** function in a loop within its **WinMain** function to remove messages from the application's message queue. This loop is called the main message loop. The **GetMessage** function searches an application's message queue and, if any messages exist, returns the top message in the queue. If the message 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 an application's **WinMain** function has retrieved a message from the application's message queue, it can dispatch the message to a window procedure by using the **DispatchMessage** function. This function directs Windows to call the window procedure of the window associated with the message, and then passes the content of the message as function arguments. The window procedure can then process the message and carry out any requested changes to the window. When the window procedure returns, Windows returns control to the main message loop in the **WinMain** function. The main message loop can then retrieve 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 message each time the user presses a key. The 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 message loop in the **WinMain** 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's message queue. The message can then be dispatched to a window procedure.

1.1.2 Translating Messages

In general, a **WinMain** 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 keyboard input is translated correctly.

The following example illustrates the typical main message loop that a **WinMain** function uses to retrieve messages from the application's message queue and dispatch them to the application's window procedures:

```
int PASCAL WinMain(hinst, hPrevInst, lpCmdLine, ShowCmd)
HINSTANCE hinst;
HINSTANCE hPrevInst;
LPSTR lpCmdLine;
int ShowCmd;
{
    MSG msg;
      .
      .
      .
while (GetMessage(&msg, NULL, 0, 0)) {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
    return msg.wParam;
}
```

An application that uses accelerator keys must load an accelerator table from the resource-definition file by using the **LoadAccelerators** function and then translate keyboard messages into accelerator-key messages by using the **Translate-Accelerator** function. For more information about accelerator keys, see the *Microsoft Windows Guide to Programming.*

The main message loop for an application that uses accelerator keys should have the following form:

```
while (GetMessage(&msg, NULL, 0, 0)) {
    if (TranslateAccelerator(hwnd, haccel, &msg) == 0) {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
     }
}
return msg.wParam;
```

The **TranslateAccelerator** function must appear before the standard **Translate-Message** and **DispatchMessage** functions. Furthermore, because **Translate-Accelerator** automatically dispatches the accelerator-key message to the appropriate window procedure, 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 to examine its message queue for specific messages without removing them from the queue. The function returns a nonzero value if a message exists in the queue and lets the application retrieve the message and process it without going through the application's main message 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 end 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 message 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 **PostAppMessage** function is a variation on **PostMessage** that posts a message using the application's module handle rather than a window handle.

The **PostMessage** function directs Windows to post a message—that is, place the message in an application's message queue. The **PostMessage** function immediately returns control 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 procedure, bypassing the application's message queue. Windows does not return control to the calling application until the window procedure that receives the message processes the message or returns control as a result of a call to the **ReplyMessage** function.

When an application transmits a message, it must do so by calling **SendMessage** if the application relies on the return value of a message. The return value of **SendMessage** is the same as the value returned by the window procedure 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 placed in the queue and does not indicate how the message was processed.

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 using the **SendMessage** function.

Typically, a task that calls **SendMessage** to send a message to another task does not continue running until the window procedure that receives the message returns. When the task that receives the message yields control, the sending task cannot continue to run and to process messages because it is waiting for **SendMessage** to return, resulting in a message deadlock.

The application processing the message does not have to yield explicitly to cause the problem. Calling any one of the following functions can result in the application yielding control:

- **DialogBox**
- **DialogBoxIndirect**
- **DialogBoxIndirectParam**
- **DialogBoxParam**

- **GetMessage**
- **MessageBox**
- **PeekMessage**
- **Yield**

Before calling any of these functions while processing a message, a window procedure should first call the **InSendMessage** function to find out whether the message was sent by the **SendMessage** function from another application. If **InSendMessage** returns a nonzero value, the window procedure must call the **ReplyMessage** function before calling any function that yields control.

1.1.6 Message Functions

Message functions read and process Windows messages in an application's message queue. Following are the message functions:

Function	Description
CallWindowProc	Passes message information to the specified window procedure.
DispatchMessage	Passes a message to the window procedure of the specified window.
GetMessage	Retrieves a message from an application's message queue.
GetMessageExtraInfo	Retrieves information about a hardware message.
GetMessagePos	Returns the position of the mouse at the time the last message was retrieved from the calling application's message queue.
GetMessageTime	Returns the time at which the last message was retrieved from the calling application's message queue.
GetQueueStatus	Returns a value that identifies the types of messages, if any, that are in the application's message queue.
hardware_event	Places a hardware message in the system queue.
InSendMessage	Finds out whether the current window procedure is processing a message that was sent as a result of another application calling the **SendMessage** function.
PeekMessage	Checks an application's message queue and returns one message from the specified range of messages, if any such messages are in the queue.
PostAppMessage	Places a message in an application's message queue.
PostMessage	Places a message in the message queue of the application associated with a specified window.
PostQuitMessage	Posts a WM_QUIT message to the calling application.

Function	Description
ReplyMessage	Replies to a message sent from a different task without returning control to the task.
SendMessage	Sends a message to a window or windows.
SetMessageQueue	Creates a new message queue of a different size.
TranslateAccelerator	Processes accelerator keys for menu commands.
TranslateMDISysAccel	Processes accelerator keystrokes for a multiple document interface (MDI) child window.
TranslateMessage	Translates virtual-keystroke messages into character messages.
WaitMessage	Yields control to other applications.
WinMain	Serves as an entry point for execution of a Windows application.

For detailed information about the message functions, see the *Microsoft Windows Programmer's Reference, Volume 2.*

1.2 Creating and Managing Windows

This section describes how to create, destroy, modify, and obtain information about windows.

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, a window class must have been created and registered for that window. An application registers a class by filling a **WNDCLASS** structure and passing a pointer to the structure 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 closes.

Although a complete window class consists of many elements, Windows requires only that an application supply a class name, the address of the window procedure that will process all messages sent to windows belonging to this class, and an instance handle identifying 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: system global classes, application global classes, and application local classes. These types differ in scope and in when and how they are created and destroyed.

1.2.1.1 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. System global classes include edit-control and list-box control classes.

1.2.1.2 Application Global Classes

An application or (more likely) a dynamic-link library (DLL) 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. Typically, a DLL creates an application global class so that applications that call the DLL can use the class. Windows destroys an application global class when the application that created it closes or the DLL that created it is unloaded. For this reason, it is essential that all applications destroy all windows using that class before the application that created the class closes or the DLL that created the class is unloaded. Use the **UnregisterClass** function to remove an application global class and free the storage associated with it.

1.2.1.3 Application Local Classes

An application local class is any window class created by an application for its exclusive use. This is the more common type of class created by an application. Use the **UnregisterClass** function to remove an application local class and free the storage associated with it.

1.2.2 How Windows Locates a Class

When an application creates a window with a specified class, Windows uses the following procedure 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, it searches the application global class list.
3. If Windows does not find the name in the application global class list, it searches the system global class list.

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

1.2.3 Class Ownership

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

1.2.4 Registering a Window Class

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

1.2.5 Shared Window Classes

An application must not share its registered classes with other applications. Some information in a window class, such as the address of the window procedure, is specific to a given application and cannot be used by other applications. However, applications can share an application global class. For more information, see Section 1.2.1.2, "Application Global Classes."

Although an application must not share one of its registered classes with other applications, 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 system-global window classes. These classes define special control windows that carry out common input tasks, such as letting the user direct scrolling, type text, 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 control windows. See the description of the **CreateWindow** function in the *Microsoft Windows Programmer's Reference, Volume 2*, for a list of the predefined window classes.

1.2.7 Elements of a Window Class

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

Element	Purpose
Class name	Distinguishes the class from other registered classes.
Window-procedure 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 or DLL that registered the class.
Class cursor	Defines the shape of the cursor when the cursor is in a window of the class.
Class icon	Defines the shape of the icon Windows displays when a window belonging to the class is minimized.
Class background brush	Defines the color and pattern Windows uses to fill the client area when the window is opened or painted. If this parameter is set to NULL, the window must paint its own background whenever it receives the WM_ERASEBKGND message.
Class menu	Specifies the default menu used for any window belonging to 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 extra memory, in bytes, that Windows should reserve at the end of the **WNDCLASS** structure. Windows initializes this memory to zero.
Window extra	Specifies the amount of extra memory, in bytes, that Windows should reserve at the end of any window structure an application creates that has this class. Windows initializes this memory to zero.

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.

1.2.7.1 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** member of the **WNDCLASS** structure to the address of a null-terminated string that specifies 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 zero, indicating failure. The 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.

1.2.7.2 Window-Procedure Address

Every class needs a window-procedure address. The address defines the entry point of the window procedure that is used to process all messages for windows in the class. Windows passes messages to the procedure when it requires the window to carry out tasks, such as painting its client area or responding to input from the user. An application assigns a window-procedure to a class by copying its address to the **lpfnWndProc** member of the **WNDCLASS** structure. The window procedure must be exported in the module-definition (.DEF) file. For more information about exporting functions, see the *Microsoft Windows Guide to Programming.*

1.2.7.3 Instance Handle

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

Multiple instances of the same application or DLL all use the same code segment, but each has its own data segment. Windows uses an instance handle to identify the data segment that corresponds to a particular instance of an application or DLL.

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

1.2.7.4 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 a predefined cursor shape by using the **LoadCursor** function, and then assigns the returned cursor handle to the **hCursor** member of the **WNDCLASS** structure. Alternatively, you can use Microsoft Image Editor (IMAGEDIT.EXE) to create your own custom cursor, and use Microsoft Windows Resource Compiler (RC) to add the cursor as a resource to your application's executable file. The application can then use the **Load-Cursor** function to load the custom cursor from the application's resources.

Windows does not require a class cursor. If an application sets the **hCursor** member of the **WNDCLASS** structure to NULL, a class cursor is not defined. Windows assumes that the window will set the cursor shape each time the cursor moves into the window. A window can set the cursor shape by calling the **SetCursor** function whenever the window receives the WM_MOUSEMOVE message.

1.2.7.5 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** member of the **WNDCLASS** structure.

Windows does not require that a window class have a class icon. If an application sets the **hIcon** member of the **WNDCLASS** structure to NULL, a class icon is not defined. In this case, Windows sends the WM_ICONERASEBKGND message to a window of the class whenever the window must paint the background of the icon. If the window does not process the WM_ICONERASEBKGND message, Windows draws an image of the contents of the window's client area onto the icon when it is minimized.

1.2.7.6 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. Windows notifies a window that its background needs to be painted by sending the WM_ERASEBKGND message to the window.

To assign a background brush to a class, an application can create a brush by using the appropriate functions from the graphics device interface (GDI) and then assign the returned brush handle to the **hbrBackground** member of the **WNDCLASS** structure.

Instead of creating a brush, an application can use a standard system color by setting the **hbrBackground** member to one of the standard system color values. For a list of the standard system color values, see the description of the **SetSysColors** function in the *Microsoft Windows Programmer's Reference, Volume 2.*

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.

1.2.7.7 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 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** member of the **WNDCLASS** structure to the address of a null-terminated string that specifies 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 application can set the **lpszMenuName** member to that integer value by applying the **MAKEINTRESOURCE** macro before assigning the value.

Windows does not require a class menu. If an application sets the **lpszMenuName** member of the **WNDCLASS** structure to NULL, 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 for a class and a child window of that class is created, the menu is ignored. For more information about menus, see Section 1.2.19, "Menus."

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. The class styles are as follows:

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. For more information about device contexts, see Section 1.2.12
CS_DBLCLKS	Sends double-click messages to the window procedure.
CS_GLOBALCLASS	Specifies that the window class is an application global class. An application global class is created by an application or DLL and is available to all applications. The class is destroyed when the application or DLL that created the class closes; it is essential, therefore, that all windows created with the application global class be closed before the application or DLL closes.
CS_HREDRAW	Requests that the entire client area be redrawn if a movement or size adjustment changes the width of the client area.
CS_NOCLOSE	Inhibits the Close command on the System menu (sometimes referred to as the Control menu).
CS_OWNDC	Allocates a unique display context for each window in the class. For more information about device contexts, see Section 1.2.12
CS_PARENTDC	Gives the parent window's display context to the child windows. For more information about device contexts, see Section 1.2.12
CS_SAVEBITS	Saves, as a bitmap, 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 that 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 size adjustment changes the height of the client area.

To assign a style to a window class, an application assigns the style value to the **style** member 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**

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 procedure (or procedures) before being passed to the class window procedure.

To create the subclass, the **SetWindowLong** function is used to change which window procedure is associated with a particular window, causing Windows to call the new window procedure instead of the previous one. An application must call the **CallWindowProc** function to pass to the previous window procedure any messages not processed by the new window procedure. This allows Windows to create a chain of window procedures. The application can retrieve the address of the previous window procedure by using the **GetWindowLong** function before using the **SetWindowLong** function.

Similarly, the **SetClassLong** function changes which window procedure is associated with a window class. Any window that is subsequently created with that class will be associated with the replacement window procedure 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 an application subclasses a window or class of windows, it must export the replacement window procedure in its module-definition file, call the **MakeProcInstance** function to create the address of the procedure, and pass the address to the **SetWindowLong** or **SetClassLong** function. For more information about module-definition files, see the *Microsoft Windows Guide to Programming.*

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 redraw the entire client area instead of copying the previous contents each time a window changes size, 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 display-context style is explicitly 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.

To avoid retrieving a display context each time it needs to paint inside 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 uses a significant amount of system resources.

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; 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. For more information about display contexts, see the *Microsoft Windows Guide to Programming.*

1.2.13 Window Procedures

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

A window procedure receives the following types of messages:

- Input messages from the keyboard, mouse or other pointing device, and timer
- Requests for information, such as a request for the window title
- Reports of changes made to the system by other windows, such as a change to the WIN.INI file
- Messages that give the window procedure an opportunity to modify the standard system response to certain actions, such as an opportunity to adjust a menu before it is displayed
- Requests to carry out some action on its window or client area, such as a request to update the client area
- Information about its status in relation to other windows, such as its losing access to the keyboard or becoming the active window

Most of the messages a window procedure 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 procedure continues to receive messages from the system and possibly other windows in the system until the window procedure, the window procedure of a parent window, or the system destroys the window. Even while the window is in the process of being destroyed, the window procedure receives additional messages that give it the opportunity to carry out any cleanup tasks before terminating. These messages include WM_CLOSE, WM_DESTROY, WM_QUERYENDSESSION, and WM_ENDSESSION. But once the window is destroyed, no more messages are passed to the procedure for that particular window. If there is more than one window of the class, however, the window procedure continues to receive messages for the other windows until they, too, are destroyed.

A window procedure defines how all windows of a given window actually behave; that is, it defines what response the windows make to commands from the user or system. The window procedure must examine messages it receives from the system and determine what action, if any, to take. For example, if the user clicks the scroll bar, the window procedure may scroll the contents of the client area. Windows passes information that affects a window and provides some tools to carry out tasks, such as drawing and scrolling, but the window procedure must carry out each actual task.

A window procedure can also choose not to respond to a given message. If it does not respond, the procedure must pass the message to the **DefWindowProc** function to give the system the opportunity to respond. 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 procedures.

A window procedure also receives messages that are really intended to be processed by the system. These messages, called nonclient-area messages, inform the procedure 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 to move or adjust the size of the window. Although Windows passes these messages to the window procedure, the procedure 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**.

1.2.13.1 Window Messages

A window message is a set of values that Windows sends to a window procedure to provide input to the window or request the window to carry out some action. Windows includes a wide variety of messages that it or applications can send to a window procedure. Most messages are sent to a window as a result of a given function being executed or as a result of input from the user.

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 procedure. The window procedure then examines the message identifier to determine what response to make and how to interpret the 16- and 32-bit values.

A window procedure must use the Pascal calling convention. The following illustrates the window procedure syntax:

LONG FAR PASCAL *WndProc*(*hwnd*, *msg*, *wParam*, *lParam*)
HWND *hwnd*;
UINT *msg*;
WPARAM *wParam*;
LPARAM *lParam*;

The *hwnd* parameter identifies the window receiving the message; the *msg* parameter is the message identifier; the *wParam* parameter is 16 bits of additional message-specific information; and *lParam* is 32 bits of additional message-specific 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 procedure, so the procedure must be declared with the **FAR** attribute. The window-procedure name must be exported by including it in an **EXPORTS** statement in the application's module-definition file.

1.2.13.2 Default Window Procedure

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

Message	Default action
WM_ACTIVATE	Activates or deactivates a window.
WM_CANCELMODE	Cancels internal processing of standard scroll bar input, cancels internal menu processing, and releases mouse capture.
WM_CHARTOITEM	Returns –1.
WM_CLOSE	Calls the **DestroyWindow** function.
WM_CTLCOLOR	Sets the background and text color and returns a handle of the brush used to fill the control background.
WM_DRAWITEM	Draws the focus rectangle for an owner-drawn list box item.
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 bytes, of the window title.
WM_ICONERASEBKGND	Fills the icon's client area with the window's background brush.

Message	Default action
WM_KEYUP	Sends a WM_SYSCOMMAND message to the top-level window if the F10 key or the ALT key was released. The *wParam* parameter of the message is set to SC_KEYMENU.
WM_MOUSEACTIVATE	Sends the WM_MOUSEACTIVATE response to the parent window. The parent determines whether to activate the child 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.
WM_NCHITTEST	Finds out what part of the window the mouse is in.
WM_NCLBUTTONDBLCLK	Tests the given point to find out the location of the mouse and, if necessary, generates additional messages.
WM_NCLBUTTONDOWN	Finds out 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 find out the location of the mouse and, if necessary, generates additional messages.
WM_NCMOUSEMOVE	Tests the given point to find out the location of the mouse and, if necessary, generates additional messages.
WM_NCPAINT	Paints the nonclient areas of the window.
WM_PAINT	Validates the current update region, but does not paint the region.
WM_QUERYENDSESSION	Returns TRUE.
WM_QUERYOPEN	Returns TRUE.
WM_SETCURSOR	Displays the appropriate mouse cursor, based on the position of the cursor.
WM_SETREDRAW	Forces an immediate update of information about the clipping region of the complete window.

Message	Default action
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.
WM_SYSKEYUP	Sends a WM_SYSCOMMAND message to the top-level window if the F10 key or the ALT key was released. The *wParam* parameter of the message is set to SC_KEYMENU.
WM_VKEYTOITEM	Returns –1.
WM_WINDOWPOSCHANGED	Sends the WM_SIZE and WM_MOVE messages to the window.
WM_WINDOWPOSCHANGING	Sends the WM_GETMINMAXINFO message to the window if the window has the WS_OVERLAPPED or WS_THICKFRAME style.

For detailed information on each Windows message, see the *Microsoft Windows Programmer's Reference, Volume 3.*

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.

1.2.14.1 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 and Maximize buttons, scroll bars, and a menu, if these items are specified when the window is created. For a window used as a main interface, the System menu and Minimize and Maximize buttons 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 restored to its previous size and position. An application minimizes a window to save screen space when several windows are open at the same time.

An application creates 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 title bar and a border. The WS_OVERLAPPEDWINDOW style creates an overlapped window with a title bar, a thick-frame border, a System menu, and Minimize and Maximize buttons. For a complete list of window styles, see the description of the **CreateWindow** function in the *Microsoft Windows Programmer's Reference, Volume 2.*

1.2.14.2 Owned Windows

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

- An owned window is always in front of its owner when the windows are in z-order. Attempting to move the owner—that is, on an imaginary z-axis extending in front of the owned window from the screen toward the user—causes the owned window also to change position to ensure that it will always be in front of 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.

1.2.14.3 Pop-up Windows

Pop-up windows are another special type of overlapped window. The main difference between a pop-up window and other overlapped windows is that an overlapped window always has a title bar, whereas the title bar is optional for a pop-up window. Like other 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. An application can use the **ShowWindow** function to open or close a pop-up window.

1.2.14.4 Child Windows

A child window is a window that is 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. An application can use the **ShowWindow** function to show or hide a child window.

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 of 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 and Maximize buttons, and scroll bars for a child window. In most cases, the application designs its own features for the child window.

Although it is not required, every child window should have a unique integer identifier. The identifier, given in the *hmenu* parameter of the **CreateWindow** function in place of a menu, helps identify the child window when its parent window has 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 multiple child windows can identify which child window is sending the message. Child windows that share the same parent window are sibling windows.

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 Chapter 2, "Graphics Device Interface.") 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, as follows:

Parent window	Child window
Shown	Shown after the parent window is shown.
Hidden	Hidden before the parent window is hidden. A child window can be visible only when the parent window is visible.
Destroyed	Destroyed before the parent window is 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 draws 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. Sibling windows can draw in each other's client area unless one child window has a WS_CLIPSIBLINGS style. If the application specifies this style for a child window, any portion of that child's sibling window that lies within this window is clipped.

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

Each window takes up system resources, so an application should not use child windows indiscriminately. For optimum performance, an application that needs to logically divide its main window should do so in the window procedure of the main window rather than by using child windows.

1.2.15 Multiple Document Interface Windows

Windows 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 that 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** structure as the function's *lpvParam* 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 about MDI, see the *Microsoft Windows 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. A title bar makes it possible for the user to move the window by using a mouse or other pointing device.

1.2.17 System Menu

The System menu, identified by a box at the left end of the title bar, is a pop-up menu that contains the system commands. (The System menu is sometimes referred to as the Control menu.) The system commands are commands that can be selected by the user to direct Windows to carry out actions that affect the window, such as moving and closing it.

To create a window with a System menu or Close box, the application must specify both the WS_SYSMENU and WS_CAPTION window styles when the window is created.

1.2.18 Scroll Bars

The horizontal and vertical scroll bars are bars on the lower and right sides of a window, respectively, making it possible for a user to 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 procedure 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. An application can use the **ShowScrollBar** function to show or hide a scroll bar of a window with the WS_HSCROLL or WS_VSCROLL style.

1.2.19 Menus

A menu is a list of commands from which the user can select using the mouse or other pointing device or the keyboard. When the user selects an item, Windows sends a corresponding message to the window procedure 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.

A pop-up menu contains a vertical list of items and is often displayed when a user selects a menu-bar item. In turn, a pop-up menu item can display another pop-up menu. A pop-up menu can float—that is, it 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**.

An application can create or modify an individual menu item with the MF_OWNERDRAW style, indicating that the item is an owner-drawn 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** structure. The owner then fills in this structure with the dimensions of the item and returns. Windows uses the information in the 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 an owner-drawn menu item. A top-level menu item cannot be an owner-drawn item.

An application can call the **AppendMenu**, **InsertMenu**, or **ModifyMenu** function to add an owner-drawn menu item to a menu or to change an existing menu item to be an owner-drawn menu item. To maintain additional data associated with the item, the application can supply a 32-bit value for the *lpNewItem* parameter of the function. This value is available to the application as the **itemData** member 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 by 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. For more information about menus, see the *Microsoft Windows Guide to Programming*.

1.2.20 Window State

The window state can be open (minimized, maximized, or restored), hidden or visible, and enabled or disabled. The initial state of a window depends on whether the following window styles are used:

- WS_DISABLED
- WS_MINIMIZE
- WS_MAXIMIZE
- WS_VISIBLE

By default, Windows creates windows that are initially enabled—that is, windows that can start receiving input messages immediately. An application can disable input to a new window 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 make it possible for the user to specify data or for the application to 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 closed. 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 procedure, the application uses the **CreateWindow** function to create the window. This function directs Windows to prepare internal 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. The **CreateWindow** function sends this message to inform the window procedure 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** structure, whose members correspond to the parameters passed to **CreateWindow**.

The WM_CREATE message is sent directly to the window procedure, bypassing the application's message queue. This means an application creates a window and processes the WM_CREATE message before it enters the main message 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 specify the window state.

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 procedure. The **DestroyWindow** function also destroys all of the window's child and owned windows.

The window procedure also receives a WM_DESTROY message when the WM_CLOSE message is processed by the **DefWindowProc** function. When a window procedure 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 close when the message is read from the queue.

1.2.22 Window-Creation Functions

Window-creation functions create, destroy, modify, and obtain information about windows. Following are the window-creation functions:

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 messages that an application-defined dialog box procedure does not process.

Function	Description
DefFrameProc	Provides default processing for messages that an application-defined MDI frame window does not process.
DefMDIChildProc	Provides default processing for messages that an application-defined MDI child window does not process.
DefWindowProc	Provides default processing for messages that an application-defined window procedure does not process.
DestroyWindow	Destroys a window.
GetClassInfo	Retrieves information about a specified class.
GetClassLong	Retrieves a long value from the extra class memory associated with a window.
GetClassName	Retrieves a window-class name.
GetClassWord	Retrieves a word value from the extra class memory associated with a window.
GetLastActivePopup	Finds out which pop-up window owned by another window was most recently active.
GetWindowLong	Retrieves a long value from the extra window memory associated with a window.
GetWindowWord	Retrieves a word value from the extra window memory associated with a window.
RegisterClass	Registers a window class.
SetClassLong	Set a long value in the extra class memory associated with a window.
SetClassWord	Set a word value in the extra class memory associated with a window.
SetWindowLong	Set a long value in the extra window memory associated with a window.
SetWindowWord	Set a word value in the extra window memory associated with a window.
UnregisterClass	Removes a window class from the window-class table.

For detailed information about the window-creation functions, see the *Microsoft Windows Programmer's Reference, Volume 2.*

1.3 Display and Movement Functions

Display and movement functions show, hide, and move windows and obtain information about the number and position of windows on the screen. Following are display and movement functions:

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	Minimizes the specified window.
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.
GetWindowPlacement	Retrieves the show state and the normal (restored), minimized, and maximized positions of a window.
GetWindowRect	Copies the dimensions of an entire window.
GetWindowText	Copies a window title into a buffer.
GetWindowTextLength	Returns the length, in bytes, of the given window's title or text.
IsIconic	Specifies whether a window is minimized (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.
SetWindowPlacement	Sets the show state and the normal (restored), minimized, and maximized positions of a window.
SetWindowPos	Changes the size, position, and ordering of overlapped, pop-up, and child windows.
SetWindowText	Sets the window title or text.
ShowOwnedPopups	Shows or hides all pop-up windows.
ShowWindow	Sets the visibility state of the given window.

For detailed information about the display and movement functions, see the *Microsoft Windows Programmer's Reference, Volume 2.*

1.4 Input Functions

Input functions disable input from system devices, take control of system devices, or define special actions that Windows takes when an application receives input from a system device. The system devices are the mouse (or other pointing device), the keyboard, and the timer. Following are input functions:

Function	Description
EnableWindow	Enables or disables mouse and keyboard input to a given window.
GetActiveWindow	Returns a handle of the active window.
GetCapture	Returns a handle of 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 has the input focus.
GetTickCount	Returns the number of timer ticks recorded since the system was started.
IsWindowEnabled	Determines whether the specified window is enabled for mouse and keyboard input.
KillTimer	Removes 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.
SwapMouseButton	Reverses the actions of the left and right mouse buttons.

For detailed information about the input functions, see the *Microsoft Windows Programmer's Reference, Volume 2.*

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. Following are hardware functions:

Function	Description
EnableHardwareInput	Enables or disables mouse and keyboard input throughout the application.
GetAsyncKeyState	Returns interrupt-level information about the key state.
GetInputState	Returns nonzero if there is mouse or keyboard input.
GetKBCodePage	Determines which code-page tables are loaded.
GetKeyboardState	Copies an array that contains the state of each key.
GetKeyNameText	Retrieves a string specifying the name of a key from a list maintained by the keyboard driver.
GetKeyState	Retrieves the state of a virtual key.
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 the ASCII values of OEM character codes 0 through 0x0FF into the OEM scan codes and shift states. For more information about the OEM character set, see the *Microsoft Windows Guide to Programming.*
SetKeyboardState	Sets the state of one or more keys by altering values in an array.
VkKeyScan	Translates a Windows character to the corresponding virtual-key code and shift state for the current keyboard.

For detailed information about the hardware functions, see the *Microsoft Windows Programmer's Reference, Volume 2.*

1.6 Painting

This section describes the system display and the preparation of windows for painting and other general-purpose graphics operations.

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 because many applications can run at one time, 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 each application has space to display output but does not draw in the space reserved for other applications.

Windows manages the system display by using display contexts. 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 painting functions described in Section 1.6.14, "Painting Functions," 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. A window display context can be used for painting within a window's nonclient area.

1.6.2.1 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 of the context. The handle can be used with GDI functions to draw in the client area of the given window. After drawing is complete, an application must use the **ReleaseDC** or **EndPaint** function to return the context to the cache. 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 the tools currently available to carry out the actual drawing. The default selections for a common display context are as follows:

Attribute	Default
Background color	Background color setting from Windows Control Panel (typically, white).
Background mode	OPAQUE.
Bitmap	No default.
Brush	WHITE_BRUSH.
Brush origin	(0,0).

Attribute	Default
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 3.0 or earlier).
Intercharacter spacing	0.
Mapping mode	MM_TEXT.
Pen	BLACK_PEN.
Polygon-filling mode	ALTERNATE.
Relative-absolute flag	ABSOLUTE.
Stretching mode	BLACKONWHITE.
Text color	Text color setting from Control Panel (typically, black).
Viewport extent	(1,1).
Viewport origin	(0,0).
Window extent	(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 more information about these functions, see the *Microsoft Windows Programmer's Reference, Volume 2.*) 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 region, are lost. Windows does not preserve the previous selections of a common display context. Applications that modify the attributes of a common display context must do so each time another context is retrieved.

1.6.2.2 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 exclusive 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 device origin and clipping region for the new window and ensures that the context is applied to the correct window. An application can use the **GetDC** or **BeginPaint** function to retrieve a handle of the class display context. The **ReleaseDC** and **EndPaint** functions have no effect on a 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. 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. 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 Chapter 2, "Graphics Device Interface."

1.6.2.3 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 exclusive use by the window. Although using private display contexts is convenient, they are expensive in terms of system resources, so an application should use them sparingly.

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. An application can use the **GetDC** or **BeginPaint** function to retrieve a handle of a private display context. The **ReleaseDC** and **EndPaint** functions have no effect on a private display context.

A private display context is given the same default selections as a common display context when the window is created. These selections can be modified at any time. Windows preserves any new selections made for the context. New selections, such as of a clipping region or 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 Chapter 2, "Graphics Device Interface."

1.6.2.4 Window Display Context

A window display context permits painting anywhere in a window, including the title 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. The 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. Because 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 can 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 to the **DefWindowProc** function. For applications that do not process WM_NCPAINT messages but still need to paint within the nonclient area, the **GetSystemMetrics** function can be used to retrieve the dimensions of various parts of the nonclient area, such as the title bar, menu bar, and scroll bars.

1.6.3 Display-Context Cache

Windows maintains a cache of display contexts that it uses for common display contexts 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 appears to be deadlocked, while the debugging version of Windows undergoes a fatal exit, alerting you 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

To manage the system display, Windows carries out many operations that affect the contents of the client area. If Windows moves, sizes, or alters the appearance of the screen, 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 of 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 can paint within 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 the 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 message 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.

To direct Windows to send a WM_PAINT message, an application can use the **UpdateWindow** function. The **UpdateWindow** function sends the message directly to the window, regardless of the number of other messages in the application's message queue. **UpdateWindow** is typically used when a window needs 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 Chapter 2, "Graphics Device Interface."

The **BeginPaint** function clears 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 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 fonts.

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** function retrieves the smallest rectangle that encloses the entire update region. The **GetUpdateRgn** function retrieves the update region itself. 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 because if it is not erased, the client area will contain whatever was originally on the screen 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 fill 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 an undesirable visual effect. For more information about brushes, see Chapter 2, "Graphics Device Interface."

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 to be the current 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. (**UnrealizeObject** should not be used on stock objects, only on brushes created by the application.)
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 screen 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, such as an application or a warning.

You can use the Image Editor to create an icon and then use Microsoft Windows Resource Compiler (RC) to add the icon to your application's resources. Your application can then call the **LoadIcon** function to load the icon into memory.

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 the icon's handle is the handle of that memory. An application can free memory used to store an icon created by **CreateIcon** by calling the **DeleteIcon** function.

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 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. For a list of the text formatting styles, see the description of the **DrawText** function in the *Microsoft Windows Programmer's Reference, Volume 2.*

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

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

DrawText recognizes a number of control characters and carries out special actions when it encounters them. The control characters and their respective actions are as follows:

Windows character	Action
Carriage return (13)	Interpreted as a line-break character. The text is immediately broken and continued on the next line down in the rectangle.

Windows character	Action
Linefeed (10)	Interpreted as a line-break character. The text is immediately broken and continued 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 wordwrap 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 wordwrap character that is closest 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 which tab-stop value is given with the DT_TABSTOP style. The default value 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 COLOR_GRAYTEXT, the solid gray system color used to draw disabled text. However, if the current 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. However, an application can intercept or modify each step of this process 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. First, **GrayString** sets text color to black. It then creates a bitmap and 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 the text is 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.

If an additional brush is combined with text, it is defined for the *hbr* parameter of **GrayString**. The brush is combined with the text as the text is copied to the client area by the **BitBlt** function. The additional brush is intended to be used to give the text a desired color, because the bitmap used to draw the text is a monochrome bitmap.

If an application-supplied function replaces **TextOut**, it is defined for the *gsprc* parameter of **GrayString**. When *gsprc* is not NULL, **GrayString** automatically calls the application-supplied function instead of the **TextOut** function and passes it a handle of the display context for the memory bitmap and 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 a nonzero value.

GrayString suppresses graying if it receives a *cch* parameter equal to –1 and the application-supplied function returns zero. This provides 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 a part of the nonclient area of the window, such as the title bar, menu bar, or window frame, needs painting. Processing this message is not recommended because a window that does so must be able to paint all the required parts of the nonclient area for the window. Unless the Windows application is creating a custom nonclient area for a child window, a window should pass this message to the **DefWindowProc** function for default processing.

1.6.14 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. Following are the painting functions:

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.

Function	Description
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. For more information about device contexts, see Section 1.2.12, "Class and Private Display Contexts," and Section 1.6.2, "Display Context Types."
GetDCEx	Retrieves the display context for the client area (as does the **GetDC** function). For more information about device contexts, see Section 1.2.12, "Class and Private Display Contexts," and Section 1.6.2, "Display Context Types."
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. For more information about device contexts, see Section 1.2.12, "Class and Private Display Contexts," and Section 1.6.2, "Display Context Types."
GrayString	Writes the characters of a string by using gray text.
InvalidateRect	Marks a rectangle for repainting.
InvalidateRgn	Marks a region for repainting.
InvertRect	Inverts the display bits of the specified rectangle.
LockWindowUpdate	Disables or reenables drawing in a window.
RedrawWindow	Updates a rectangle or region within a window's client area.
ReleaseDC	Releases a display context. For more information about device contexts, see Section 1.2.12, "Class and Private Display Contexts," and Section 1.6.2, "Display Context Types."
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.

For detailed information about the painting functions, see the *Microsoft Windows Programmer's Reference, Volume 2.*

1.7 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.

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. A dialog box is convenient to use because all aspects of the dialog box, except how to carry out its tasks, are predefined. A dialog box supplies a window class and procedure; the window for the dialog box is created automatically. The application supplies a dialog box procedure to carry out tasks and a dialog box template that describes the dialog box style and content. For additional information about dialog boxes, see the *Microsoft Windows Guide to Programming*.

1.7.1.1 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. A modeless dialog box makes it possible for a user to supply more than one piece of information about the current task without having to select a command from a menu each time. For example, a modeless dialog box is 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 box procedure of a modeless dialog box must send a message to the parent window when it has input for the parent window. The dialog box procedure must also destroy the dialog box when it is no longer needed. An application can call the **DestroyWindow** function to destroy a modeless dialog box. The application must not call the **EndDialog** function to destroy a modeless dialog box.

1.7.1.2 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.

A modal dialog box disables its parent window, and it creates its own message loop, temporarily taking control of the application's message queue from the application's main message loop.

By default, a modal dialog box cannot be moved by the user. An application can create a movable modal dialog box by specifying the WS_CAPTION window style.

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

1.7.1.3 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 because they effectively shut down the system until the user supplies the required information.

1.7.2 Creating a Dialog Box

A dialog box is typically 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.

1.7.2.1 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. You use the Resource Compiler to convert the text description of the template to the required binary form and to add that binary form to the application's executable file.

Because a dialog box is system-independent, you can easily modify the template without changing the source code.

The **CreateDialog** or **DialogBox** function loads the resource into memory when it creates the dialog box and then uses the information in the dialog box template to create the dialog box, position it, and create and position the controls for the dialog box.

1.7.2.2 Dialog Box Measurements

Dialog box and control dimensions and coordinates are device-independent. Because 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. This ensures the best possible appearance of characters. One unit in the x-direction is equal to one-fourth of the dialog box base width unit. One unit in the y-direction is equal to one-eighth of the dialog box base height unit. The dialog box base units are computed from the height and width of the system font; the **GetDialogBaseUnits** function returns the dialog box 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, and it does not allow the width of a dialog box 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 box procedure has called the **EndDialog** function to signal the destruction 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 because they do not use the **EndDialog** function to close and do not return control in the same way a modal dialog box does. Instead, a modeless dialog box returns values to its parent window 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. A modal dialog box can also use this technique to return values to its parent window before closing.

1.7.4 Controls in a Dialog Box

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. A dialog box can contain any number and any types of controls. 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.

1.7.4.1 Control Identifiers

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

Each control in the dialog box must have a unique identifier. If a dialog box has a menu bar, there must be no conflict between menu-item identifiers and control identifiers. Because Windows sends menu input to a dialog box procedure as WM_COMMAND messages, conflicts with menu and control identifiers can cause errors. Menus in dialog boxes are not recommended.

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

1.7.4.2 The WS_TABSTOP and WS_GROUP Control Styles

The WS_TABSTOP style specifies that the user can move the input focus to the given control by pressing the TAB key 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 keystrokes. 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 within a group of controls by using the arrow keys. The first control in a group of controls must have the WS_GROUP style. The next control that has the WS_GROUP style marks the bottom boundary of the group; the input focus cannot be moved to this control by using the arrow keys. The DOWN ARROW and RIGHT ARROW keys move the input focus to controls in the order in which they have been created. The UP ARROW and LEFT ARROW keys move the input focus in reverse order. For modal dialog boxes, the arrow 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 keystrokes. Otherwise, the keys have no special meaning and the WS_GROUP style is ignored.

1.7.4.3 Buttons

Buttons are the principal interface of a dialog box. Almost all dialog boxes have at least one push button, and most have one default push button (a push button having the BS_DEFPUSHBUTTON style) and one or more other push buttons. Many dialog boxes have collections of radio buttons enclosed in group boxes or have 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 IDOK so that the action the dialog box procedure takes when the button is chosen 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. Most dialog boxes that use the special keyboard interface can also set the control identifier of another push button to IDCANCEL so that the action of the ESC key is duplicated by choosing that button.

When a dialog box first starts, the dialog box procedure can set the initial state of each button 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 box procedure can use the **CheckRadioButton** function to set the appropriate radio button and automatically clear any other radio button.

Before a dialog box terminates, the dialog box procedure 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.

1.7.4.4 Edit Controls

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

Because edit controls do not automatically return their text to the dialog box, the dialog box procedure 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 box procedure 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 box procedure can retrieve a number from an edit control by using the **GetDlgItemInt** function, which retrieves the text from the edit 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 box procedure can display an integer by using the **SetDlgItemInt** function. **SetDlgItemInt** converts a signed or unsigned integer to a string of decimal digits.

1.7.4.5 List Boxes and Directory Listings

Some dialog boxes display lists, such as a list of filenames, from which the user can select one or more items. To display a list of filenames, a dialog box typically uses a list box 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, these two functions provide a convenient way for a dialog box to display a directory listing that makes it possible for the user to select a file without having to type the location and name of the file.

1.7.4.6 Combo Boxes

Another method for providing a list of items to a user is by using a combo box. A combo box consists of either a static control or edit control 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 control, that control always displays the current selection (if any) from the list box portion of the combo box. If the combo box uses an edit control, the user can type a selection; the list box highlights the first item (if any) that matches what the user has entered in the edit control. The user can choose the OK button or press ENTER to complete the choice.

1.7.4.7 Owner-Drawn Dialog Box Controls

List boxes, combo boxes, and buttons can be designated as owner-drawn controls by creating them with the appropriate style. Following are available styles:

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

When a control has the owner-drawn style, Windows handles the user's interaction with the control as usual, performing such tasks as detecting when a user has chosen a button and notifying the button's owner of the event. However, because the control is owner-drawn, the owner of the control is completely responsible for the visual appearance of the control. Owner-drawn list boxes and combo boxes can control the display of only the individual elements within a list box or combo box, not the entire list box or combo box.

When Windows first creates a dialog box containing owner-drawn controls, it sends the owner a WM_MEASUREITEM message for each owner-drawn control. The *lParam* parameter of this message contains a pointer to a **MEASUREITEM-STRUCT** structure. When the owner receives the message for a control, the owner fills in the appropriate members 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, the WM_MEASUREITEM message is sent to the owner for each item in the control, because each item can differ in height. Otherwise, this message is sent once for the entire owner-drawn control.

Whenever an owner-drawn 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** 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 **DELETEITEMSTRUCT** structure that describes the deleted item.

1.7.4.8 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 box procedure 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 Keyboard Interface for Dialog Boxes

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 change the input focus from one control to another. The keys used in this interface and the respective actions are as follows:

Key	Action
DOWN ARROW	Moves the input focus to the next control in the group.
ENTER	Sends a WM_COMMAND message to the dialog box procedure. The *wParam* parameter is set to 1 or the default button.
ESC	Sends a WM_COMMAND message to the dialog box procedure. The *wParam* parameter is set to 2.

Key	Action
LEFT ARROW	Moves the input focus to the previous control in the group.
RIGHT ARROW	Moves the input focus to the next control in the group.
SHIFT+TAB	Moves the input focus to the previous control that has the WS_TABSTOP style.
TAB	Moves the input focus to the next control that has the WS_TABSTOP style.
UP ARROW	Moves the input focus to the previous control in the group.

The TAB key and the arrow 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 accelerator keys 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 need those boxes to have the special keyboard interface must filter all messages retrieved from the application's message queue through the **IsDialogMessage** function before carrying out any other processing. This means that the application must pass the message to **IsDialogMessage** 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 ALT+application-defined mnemonic key sequences.

In modal dialog boxes, the arrow 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 arrow 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.

1.7.6 Functions for Dialog Boxes

The functions listed in this section create, alter, test, and destroy dialog boxes and controls within dialog boxes. Following are the functions for dialog boxes:

Function	Description
CheckDlgButton	Places or removes a check mark, or changes the state of a three-state button or check box.
CheckRadioButton	Selects a specified radio button and clears 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 then passes data to it.
CreateDialogParam	Creates a modeless dialog box and then passes data to it.
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 then passes data to it.
DialogBoxParam	Creates a modal dialog box and then passes data to it.
DlgDirList	Fills a list box with names of files matching a path.
DlgDirListComboBox	Fills a combo box with names of files matching a specified path and filename.
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 identifier of a control window.
GetDlgItem	Retrieves the handle of a dialog box control in the given dialog box.
GetDlgItemInt	Translates the control text of a control into an integer value.
GetDlgItemText	Copies a control's text into a string.
GetNextDlgGroupItem	Returns the window handle of the next item in a group.

Function	Description
GetNextDlgTabItem	Returns the window handle of the next or previous item.
IsDialogMessage	Determines whether a message is intended for the given modeless dialog box.
IsDlgButtonChecked	Tests whether a button is selected.
MapDialogRect	Converts the dialog box coordinates to client coordinates.
SendDlgItemMessage	Sends a message to a control within a dialog box.
SetDlgItemInt	Sets the title or text of a control to a string that represents an integer.
SetDlgItemText	Sets the title or text of a control to a string.

For detailed information about the functions for dialog boxes, see the *Microsoft Windows Programmer's Reference, Volume 2.*

1.8 Scrolling

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.

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 scroll bar, if used, always appears at the right of the client area; a horizontal scroll bar, if used, 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 description of the **GetSystemMetrics** function in the *Microsoft Windows Programmer's Reference, Volume 2.*)

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 that it has the focus by displaying a flashing caret in the scroll box (also called 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 spectrum of colors.

1.8.2 Scroll Box

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

A scroll box position is represented as 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 through 100, there are 101 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 through 100; scroll-bar controls have an empty range (both minimum and maximum values are 0) if no explicit range is given when the control is created. An application can change the range by using the **SetScrollRange** function to set 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 through 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 scroll box along the scroll bar.

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 messages' *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 scrolling; for example, the user may scroll up one line, scroll down a page, or scroll to the bottom. The type of scrolling is determined by which area of the scroll bar the user clicks.

The user can also make a scrolling request by using the scroll box, the small rectangle inside the scroll bar. The user moves the scroll box by moving the mouse while holding the left mouse button down when the cursor is positioned on the scroll box. 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 scroll box. Each message specifies the current position of the scroll box.

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 it needs a window procedure to process the WM_HSCROLL and WM_VSCROLL messages that represent the scrolling requests. Although the result of a scrolling request depends entirely on how the window processes it, a window typically carries out a scroll operation by moving through the application's displayed information 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, which require completely new contents.

For some requests, such as SB_LINEUP and SB_LINEDOWN, not all the contents need to be erased, since some are still 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 is 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 keeps 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 hidden and, in effect, disabled. Using this technique, you can 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 equal minimum and maximum values are specified. 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 scroll box's position.

1.8.7 Scrolling Functions

Scrolling functions control the scrolling of a window's contents and control the window's scroll bars. Following are the scrolling functions:

Function	Description
EnableScrollBar	Enables or disables one or both arrows of a scroll bar.
GetScrollPos	Retrieves the current position of the scroll box.
GetScrollRange	Copies the minimum and maximum scroll-bar positions for given the scroll bars for a specified scroll operation.
ScrollDC	Scrolls a rectangle of bits horizontally and vertically.
ScrollWindow	Moves the contents of the client area.
ScrollWindowEx	Moves the contents of the client area (as does the **ScrollWindow** function) but with extended capabilities.
SetScrollPos	Sets the scroll box.
SetScrollRange	Sets the minimum and maximum scroll-bar positions.
ShowScrollBar	Displays or hides a scroll bar and its controls.

For detailed information about the scrolling functions, see the *Microsoft Windows Programmer's Reference, Volume 2.*

1.9 Menu Functions

A menu is an input tool in a Windows application that offers users one or more items, 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.

Menu functions create, modify, and destroy menus. Following are the menu functions:

Function	Description
AppendMenu	Appends a menu item to a menu.
CheckMenuItem	Places or removes check marks 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 of the menu of a specified window.
GetMenuCheckMarkDimensions	Retrieves the dimensions of the default menu check-mark 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.
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.
IsMenu	Determines if a menu handle is valid.
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 whether an item is or is not checked.
TrackPopupMenu	Displays a pop-up menu at a specified screen location and tracks user interaction with the menu.

For detailed information about the menu functions, see the *Microsoft Windows Programmer's Reference, Volume 2*.

1.10 Information Functions

Information functions obtain information about the number and position of windows on the screen. Following are the information functions:

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 title.
GetNextWindow	Returns a handle of the next or previous window.
GetParent	Retrieves the handle of the specified window's parent window.
GetTopWindow	Returns a handle of the top-level child window.
GetWindow	Returns a handle of a window that has the specified relationship to the given window.
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.
SystemParametersInfo	Retrieves or sets systemwide values.
WindowFromPoint	Identifies the window containing a specified point.

For detailed information about information functions, see the *Microsoft Windows Programmer's Reference, Volume 2.*

1.11 System Functions

System functions return information about the system metrics, color, and time. Following are the system functions:

Function	Description
GetCurrentTime	Returns the time elapsed since the system was started.
GetSysColor	Retrieves the system color.

Function	Description
GetSystemMetrics	Retrieves information about the system metrics.
GetTimerResolution	Retrieves the timer resolution.
SetSysColors	Changes one or more system colors.
SystemParametersInfo	Queries or sets systemwide parameters.

For detailed information about system functions, see the *Microsoft Windows Programmer's Reference, Volume 2.*

1.12 Clipboard Functions

The clipboard provides a mechanism that makes it possible for applications to pass data handles to other applications. For more information about the clipboard, see the *Microsoft Windows Guide to Programming.*

Clipboard functions carry out data interchange between Windows applications. Following are the clipboard functions:

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.
GetOpenClipboardWindow	Retrieves the handle of the window that currently has the clipboard open.
GetPriorityClipboardFormat	Retrieves data from the clipboard in the first format in a prioritized format list.
IsClipboardFormatAvailable	Returns nonzero if the data in the given format is available.

Function	Description
OpenClipboard	Opens the clipboard.
RegisterClipboardFormat	Registers a new clipboard format.
SetClipboardData	Copies a handle of data for the clipboard.
SetClipboardViewer	Adds a handle to the clipboard-viewer chain.

For detailed information about clipboard functions, see the *Microsoft Windows Programmer's Reference, Volume 2.*

1.13 Error Functions

Error functions display error messages and prompt the user for a response. Following are the error functions:

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

For detailed information about error functions, see the *Microsoft Windows Programmer's Reference, Volume 2.*

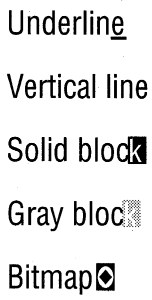
1.14 The Caret

The Windows caret 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.

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, width, and height. Windows flashes the caret by alternately inverting the display and restoring it to its previous appearance. The caret's flash rate, 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 vary in color and shape; a bitmap caret can be given any pattern. The following illustration shows some typical variations in the appearance of the caret.



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. All applications must cooperatively share the caret. Because Windows does not inform an application when a caret is created or destroyed, each window should create, move, show, or 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.

Your application can use the **CreateBitmap** function to create a bitmap for the caret; or, after you have used the Image Editor to create a bitmap and have used the Resource Compiler to add it to your application's resources, your application can use the **LoadBitmap** function to load the bitmap from the application's resources.

1.14.3 Caret Functions

Caret functions create, destroy, display, and hide the caret and alter its blink time. Following are the caret functions:

Function	Description
CreateCaret	Creates a caret.
DestroyCaret	Destroys the current caret.

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

For detailed information about the caret functions, see the *Microsoft Windows Programmer's Reference, Volume 2.*

1.15 The Cursor

The cursor is a bitmap, displayed on the screen. The user can use a mouse or other pointing device to move this bitmap to an item on the screen, such as a window or an icon. (In the remainder of section, the term mouse is used for any pointing device.)

1.15.1 The Mouse and the Cursor

When a system has a mouse, 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. For an introduction to the cursor functions, see Section 1.15.6, "Cursor Functions."

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 the **SetCursor** function, the cursor remains on the screen until a subsequent call to **SetCursor** with the parameter set to NULL removes the cursor, or until a system command is carried out. Applications that need to use the cursor without a mouse usually simulate mouse input by using keys, such as the arrow 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. Showing the cursor increments the counter; hiding the cursor decrements the counter. The cursor is only visible when the count is not a negative value.

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 any mouse movement causes the cursor to be redrawn at the mouse location. The **SetCursorPos** and **GetCursorPos** functions are most often used in applications that use the keyboard and specified keystrokes to move the cursor. Note that screen coordinates are not affected by the mapping mode in a window's client area.

1.15.4 The Cursor Hot Spot and Confining the Cursor

The hot spot of the cursor is the location in the cursor bitmap that is tracked and recognized as the position of the mouse or keyboard arrow key. For example, the hot spot on the pointer is the point at the tip of the arrow.

The **ClipCursor** function confines the cursor to a given rectangle on the 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 active 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. Windows 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.

1.15.6 Cursor Functions

Cursor functions set, move, show, hide, and confine the cursor. Following are the cursor functions:

Function	Description
ClipCursor	Restricts the cursor to a given rectangle.
CopyCursor	Copies a cursor.
CreateCursor	Creates a cursor from two bit masks.
DestroyCursor	Destroys a cursor created by the **CreateCursor** function.
GetClipCursor	Retrieves the screen coordinates of the rectangle to which the cursor has been restricted.
GetCursor	Retrieves the handle of the current cursor.
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.

For detailed information about the cursor functions, see the *Microsoft Windows Programmer's Reference, Volume 2.*

1.16 Hooks

A hook is a point in the Windows message-handling mechanism that an application can use to gain access to the message stream. Windows provides many types of hooks; each type provides access to a particular type or range of messages. To take advantage of a particular hook, an application can install a filter function that processes the messages associated with the hook. A filter function processes the messages before they reach the destination window procedure.

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. An application passes a filter function to a system hook with a call to the **SetWindowsHook** function. Each call adds a new filter function to the beginning of the chain. Whenever an application passes the address of a filter function to a system hook, it must reserve space for the address of the next filter function in the chain. **SetWindowsHook** installs a hook function into a hook chain and returns a handle of the hook.

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.

The standard window hooks and debugging hooks are as follows:

Type	Purpose
WH_CALLWNDPROC	Installs a window filter.
WH_CBT	Installs a computer-based training (CBT) filter.
WH_DEBUG	Installs a debugging filter.
WH_GETMESSAGE	Installs a message filter (on debugging versions only).
WH_HARDWARE	Installs a nonstandard hardware-message filter.
WH_JOURNALPLAYBACK	Installs a journaling playback filter.
WH_JOURNALRECORD	Installs a journaling record filter.
WH_KEYBOARD	Installs a keyboard filter.
WH_MOUSE	Installs a mouse-message filter.
WH_MSGFILTER	Installs a message filter.
WH_SYSMSGFILTER	Installs a systemwide 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 (.DEF) file.
2. Obtain the function's address by using the **GetProcAddress** function. (The **MakeProcInstance** function is used only when the filter function is not in a DLL.)
3. Call the **SetWindowsHook** function, specifying the type of hook function and the address of the function (returned by **GetProcAddress**).
4. Store the return value from **SetWindowsHook** in a reserved location. This value is the handle of the previous filter function.

Note Filter functions must reside in fixed library code and data. This allows hooks to operate in a large-frame Expanded Memory Specification (EMS) environment.

1.16.3 Hook Functions

Following are the hook functions:

Function	Description
CallMsgFilter	Passes a message and other data to the filter function for the current message.
CallNextHookEx	Passes hook information down the hook chain.
DefHookProc	Calls the next filter function in a filter-function chain.
SetWindowsHookEx	Installs a system filter function, an application filter function, or both.
UnhookWindowsHookEx	Removes a Windows filter function from a filter-function chain.

For detailed information about the hook functions, see the *Microsoft Windows Programmer's Reference, Volume 2.*

1.17 Property Lists

A property list is a storage area that contains handles for data that the application needs to associate with a window.

1.17.1 Using Property Lists

Once a data handle is in a window's property list, any application that can access the window can also access the handle. Using the property list is a convenient way to make data (for example, an alternate title or menu for a window) available when the application needs to modify a window.

Every window has its own property list. When a window is created, the list is empty. The **SetProp** function adds entries to the list. Each entry contains a unique

Windows character string and a data handle. The Windows character string identifies the handle; the handle identifies the data associated with the window, as shown in the following illustration.

Windows string	**Handle**
"binary data"	*hMemory*
"icon"	*hicon*
"screen text"	*hText*
⋮	⋮

The data handle can identify any object that the application needs 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 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 can use them. An application can use the Windows clipboard to share global memory handles with other applications. (For more information about the clipboard, see the *Microsoft Windows Guide to Programming.*) 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 then carry out the necessary 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 allocated in 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 number of entries depends on how much space is available in the USER heap. The available space 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 closes, 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.

An application can use the **RemoveProp** function at any time to remove entries from the property list. 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 that Windows has added to the list.

1.17.2 Property Functions

Property functions create and access a window's property list. Following are the property functions:

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

For detailed information about the property functions, see the *Microsoft Windows Programmer's Reference, Volume 2.*

1.18 Rectangles

In Windows, a rectangle is defined by a **RECT** structure. The structure specifies 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.

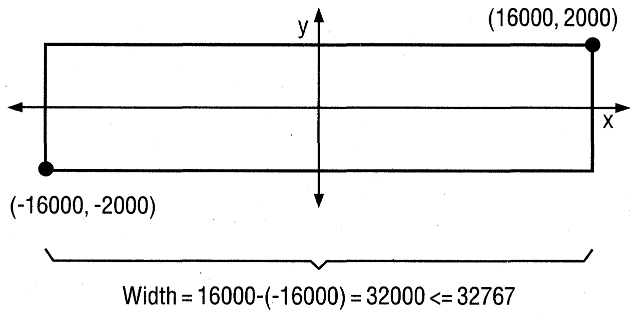
1.18.1 Using Rectangles in a Windows Application

Rectangles are used to specify rectangular areas on the screen or in a window, such as the cursor clipping region, 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.

Because 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 units of measure are determined by the function in which the rectangle is used.

1.18.2 Rectangle Coordinates

Valid coordinate values for a rectangle are in the range –32,768 through 32,767. Valid widths and heights, which must be positive, are in the range 0 through 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. Following is a rectangle whose upper-left corner is left of the origin and whose width is less than 32,767.



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.

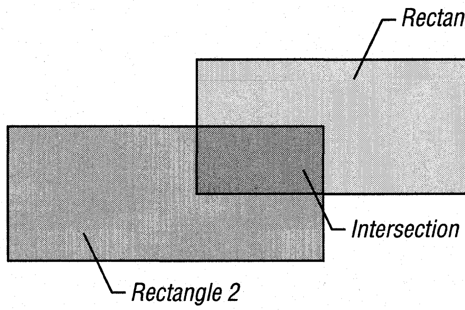
The **InflateRect** function increases or decreases the width or height of a rectangle, or both. It can add or remove width from both ends of the rectangle; it can add or remove height from both the top and bottom of the rectangle.

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

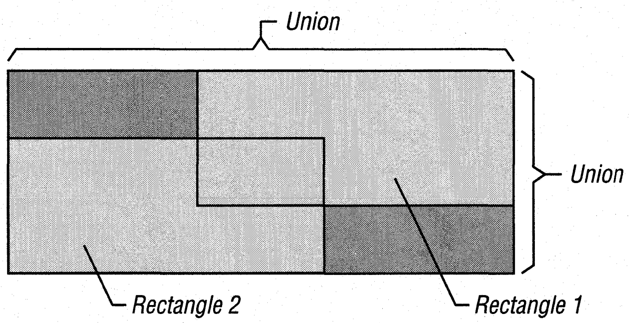
The **PtInRect** function finds out whether a given point lies within a given rectangle. The point is in the rectangle if it lies on the left or top side or is completely within the rectangle.

The **IsRectEmpty** function finds out 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 can be illustrated as follows.



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 can be illustrated as follows.



For information about functions that draw ellipses and polygons, see Chapter 2, "Graphics Device Interface."

1.18.4 Rectangle Functions

Rectangle functions alter and obtain information about rectangles in a window's client area. Following are the rectangle functions:

Function	Description
CopyRect	Makes a copy of an existing rectangle.
EqualRect	Finds out whether two rectangles are equal.
GetBoundsRect	Returns current accumulated bounding rectangle.
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.
SetBoundsRect	Controls bounding-rectangle accumulation.
SetRectEmpty	Sets a rectangle to an empty rectangle.
SubtractRect	Creates a rectangle from the difference between two rectangles.
UnionRect	Stores the union of two rectangles.

For detailed information about the rectangle functions, see the *Microsoft Windows Programmer's Reference, Volume 2.*

1.19 Related Topics

For more information about window management functions, see the *Microsoft Windows Programmer's Reference, Volume 2.*

For more information about Windows data types, messages, structures, and macros, see the *Microsoft Windows Programmer's Reference, Volume 3.*

For general information about developing Windows applications, see the *Microsoft Windows Guide to Programming.*

For information about creating, editing, and compiling resources for Windows applications, see *Microsoft Windows Programming Tools.*

Graphics Device Interface

Chapter 2

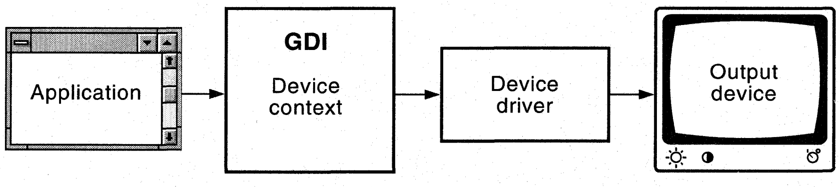
This chapter describes the functions that perform device-independent graphics operations in an application for the Microsoft Windows operating system. These operations include the creation of line, text, and bitmap output on different output devices. The functions performing those operations constitute the Windows graphics device interface (GDI).

Some Windows functions in the USER application programming interface (API) are closely related to these GDI function groups. For a full description of these functions, see Chapter 1, "Window Management."

2.1 Device Contexts

A device context (DC) is a link between a Windows application, a device driver, and an output device, such as a printer or plotter. Windows maintains a cache of five special device contexts for the system display. Applications must release these device contexts after using them.

The following illustration shows the flow of information from a Windows application through a device context and a device driver to an output device.



2.1.1 Accessing Output Devices

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.

2.1.1.1 Saving and Restoring a Device Context

The **SaveDC** and **RestoreDC** functions save and restore device contexts. The former saves the original attributes, and the latter makes them available at a later time. For example, a Windows application may need to save its original clipping region so that it can restore the original state of the client area after a series of alterations occur.

2.1.1.2 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. **DeleteDC** should be used to delete device contexts created by the application. If the application uses the **GetDC** function to retrieve a device context, it should use the **ReleaseDC** function, not **DeleteDC**.

2.1.1.3 Creating a Compatible Device Context

The **CreateCompatibleDC** function causes Windows to treat a portion of memory as a virtual device. Then Windows prepares a device context that has the same attributes as the device for which the virtual device 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 the application sends to the device is drawn in the selected bitmap. Because the device context is compatible with an 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 The **CreateCompatibleDC** function works only for devices that support raster operations. To discover whether a device supports raster operations, an application can call the **GetDeviceCaps** function with the RC_BITBLT index.

2.1.1.4 Creating an Information Context

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

2.1.2 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 structure that contains the device-context attributes is called the device-context data block. The default attributes and the GDI functions that affect or use them are as follows.

Attribute	Default	GDI functions
Background color	White	**SetBkColor**
Background mode	OPAQUE	**SetBkMode**
Bitmap	No default	**CreateBitmap** **CreateBitmapIndirect** **CreateCompatibleBitmap** **SelectObject**
Brush	WHITE_BRUSH	**CreateBrushIndirect** **CreateDIBPatternBrush** **CreateHatchBrush** **CreatePatternBrush** **CreateSolidBrush** **SelectObject**
Brush origin	(0,0)	**SetBrushOrg** **UnrealizeObject**
Clipping region	Display surface	**CreateEllipticRgn** **CreateEllipticRgnIndirect** **CreatePolygonRgn** **CreatePolyPolygonRgn** **CreateRectRgn** **CreateRoundRectRgn** **ExcludeClipRect** **IntersectClipRect** **OffsetClipRgn** **SelectClipRgn**

Attribute	Default	GDI functions
Color palette	DEFAULT_PALETTE	**CreatePalette** **RealizePalette** **SelectPalette** **UnrealizeObject**
Current pen position	(0,0)	**LineTo** **MoveTo**
Drawing mode	R2_COPYPEN	**SetROP2**
Font	SYSTEM_FONT	**CreateFont** **CreateFontIndirect** **SelectObject**
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**

2.1.3 Device-Context Functions

Device-context functions create, delete, and restore device contexts. Following are the GDI device-context functions:

Function	Description
CreateCompatibleDC	Creates a memory device context.
CreateDC	Creates a device context.
CreateIC	Creates an information context.
DeleteDC	Deletes a device context.

Function	Description
GetDCOrg	Retrieves the origin of a specified device context.
ResetDC	Updates a device context.
RestoreDC	Restores a device context.
SaveDC	Saves the current state of the device context.

In addition, applications can use the following functions in the USER API to retrieve and release device contexts:

Function	Description
BeginPaint	Prepares a window for painting, fills a buffer with information about the painting, and retrieves a handle of a device context.
GetDC	Retrieves the handle of a device context for the client area of the given window.
GetWindowDC	Retrieves a device context for an entire window, including title bar, menus, and scroll bars.
ReleaseDC	Releases a device context, freeing it for use by other applications.

For more information about these USER functions, see Chapter 1, "Window Management."

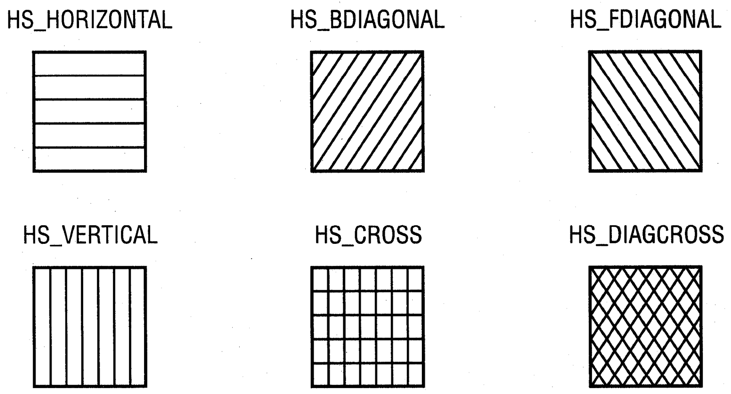
2.2 Drawing Tools

A Windows application can use drawing 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 interior of the region or object 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.)

2.2.1 Using Brushes

There are six predefined brushes available in GDI: black, dark-gray, gray, hollow, light-gray, null, and white. (Hollow and null brushes are identical.) An application selects any one of them by using the **GetStockObject** function.

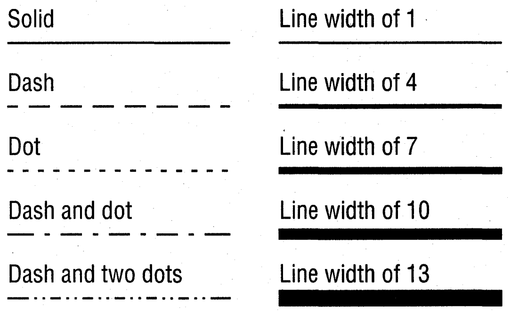
There are six hatched brush patterns: backward diagonal, cross, diagonal cross, forward diagonal, horizontal, and vertical. (A hatch line is a thin line that appears at regular intervals on a solid background.) An application can select any one of the six patterns by using the **CreateHatchBrush** function. The following illustration shows the different hatched brush patterns.



2.2.2 Using Pens

There are three predefined pens available in GDI: black, null, or white. An application selects any one of them by using the **GetStockObject** function.

An application can create an original pen by using the **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, or dotted; it can combine an alternating dot and dash or two dots and a dash; or it can be null. The pen width is the number of logical units GDI maps to a specific 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 (red, green, blue) color value. The following figure shows a variety of pen patterns obtained from calls to **CreatePen**:



2.2.3 Specifying Colors

Many of the GDI functions that create pens and brushes require that the calling application specify a color in the form of a doubleword. The color can be specified as:

- An explicit RGB value
- An index to a logical-palette entry
- A palette-relative RGB value

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

An explicit RGB doubleword 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, the third byte contains the blue field, and 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 doubleword 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 logical color closest 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 selects a solid color. If the device uses color raster technology and the RGB value specifies a color for a brush, the driver selects from a variety of available color combinations. Because many color devices can display only a few colors, the actual color is simulated by dithering (that is, mixing pixels of colors that the device can actually render).

If the device is monochrome (black-and-white), the driver selects 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** macros extract the values for red, green, and blue from an explicit RGB doubleword value.

2.2.4 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. Following are the drawing-tool functions:

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.
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.
GetBrushOrgEx	Retrieves the origin of the current brush.
GetObject	Copies the bytes of logical data that define an object.
GetStockObject	Retrieves a handle of one of the predefined stock pens, brushes, fonts, or color palettes.
IsGDIObject	Determines if handle is not GDI object.
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.

2.3 Color Palettes

Many color graphics displays are capable of displaying a wide range of colors. In most cases, however, the actual number of colors that 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.

To render colors, a 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. When a window has the input focus, Windows ensures that the window displays 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 process significantly reduces undesirable changes in the colors displayed in inactive windows.

2.3.1 Understanding Color Palettes

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

An application employs the system palette by creating and using one or more logical palettes. Each entry in the system 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 the logical palette.

Because 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 the applications.

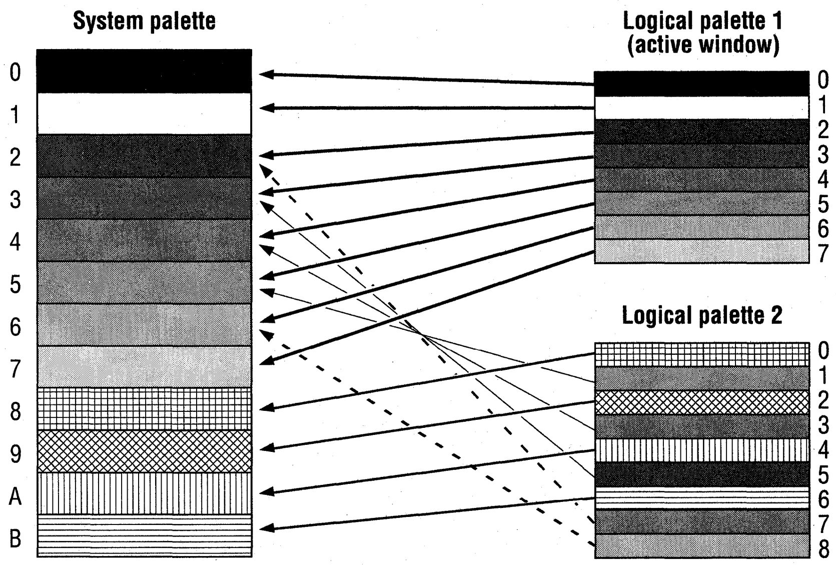
When a window requests that its logical palette be given its requested colors (a process known as realizing its palette), Windows first matches entries in the logical palette to current entries in the system palette. If an exact match for a given logical palette entry is not possible, Windows sets the entry in the logical palette into an unused entry in the system palette.

When all entries in the system palette have been used, Windows takes the 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 color matching, Windows sets aside 20 static colors in the system palette (the default palette) to which it can match entries in a background palette.

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

The following illustration shows this process. In this illustration, 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. 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.

Logical Palette 2 is owned by a window that realized its logical palette while it was inactive. Three of the colors (1, 3, and 5) in Logical Palette 2 were identical to colors in the system palette. To save space in the palette, Windows simply matched those colors to existing system colors when the second application realized its palette. Colors 0, 2, 4, and 6 were not already in the system palette, however, so Windows mapped those colors into the system palette. Because the system palette became full, Windows was not able to map the remaining two colors (which did 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.



Palette Manager Color-Mapping Algorithm

2.3.2 Using a Color Palette

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

All functions with 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 specifies 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 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 value. If the target display device supports logical palettes, Win-dows matches the palette-relative RGB value to the closest palette entry. If the target device does not support palettes, the RGB value is used as though it were an explicit RGB value. The palette-relative RGB value is identical to an explicit RGB value except that the second bit of the high-order byte is set to 1. For example, 0x02FF0000 specifies a palette-relative RGB value for pure blue. The **PALETTERGB** macro accepts values for red, green, and blue and returns a palette-relative RGB value, which an application can use as a parameter for GDI functions that require a color.

If an application specifies an RGB value instead of a palette entry, Windows uses the closest matching color in the default palette of 20 static colors.

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** function with the DIB_RGB_COLORS flag to retrieve the bitmap bits from the source bitmap in a device-independent format. Then you use the **SetDIBits** function to set the retrieved bits in the destination bitmap. This ensures that Windows properly matches colors between the two device contexts.

Note The **BitBlt** function successfully moves 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.

2.3.3 Color-Palette Functions

Windows color palettes allow an application to use as many colors as needed without interfering with its own color display or colors displayed by other windows. Following are 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.
GetNearestColor	Retrieves the solid color closest to a specified logical color that a given device can represent.
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.
ResizePalette	Changes the size of the specified logical palette.
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 process allows an inactive window to correct its colors without redrawing its client area.

The USER API also provides two palette-management functions:

Function	Description
RealizePalette	Maps entries in a logical palette to the system palette.
SelectPalette	Selects a logical palette into a device context.

For more information about these USER functions, see Chapter 1, "Window Management."

2.4 Drawing Attributes

A drawing attribute can take one of the following forms: line, brush, text, or bitmap output.

2.4.1 Setting Colors

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 set the background mode and color.

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.

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 cell), and the **SetTextColor** function sets the color of the character itself.

2.4.2 Controlling Stretch

The appearance of bitmap output can be affected by the stretch mode, which determines how lines eliminated from the bitmap are combined. 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 the **SetStretchBltMode** function to set the current stretch mode for a device context.

2.4.3 Drawing-Attribute Functions

Drawing-attribute functions affect the appearance of Windows output. Following are the drawing-attribute functions:

Function	Description
GetBkColor	Returns the current background color.
GetBkMode	Returns the current background mode.
GetPolyFillMode	Retrieves the current polygon-filling mode.

Function	Description
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.

2.5 Mapping Modes

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.

There are eight different GDI mapping modes, each of which has a specific use in a Windows application. Following are these mapping modes:

Mapping mode	Description
MM_ANISOTROPIC	Maps one logical unit to an arbitrary physical unit. The x-axis and y-axis are arbitrarily scaled.
MM_HIENGLISH	Maps one logical unit to 0.001 inch. The positive y-axis extends upward.
MM_HIMETRIC	Maps one logical unit to 0.01 millimeter. The positive y-axis extends upward.
MM_ISOTROPIC	Maps 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	Maps one logical unit to 0.01 inch. The positive y-axis extends upward.
MM_LOMETRIC	Maps one logical unit to 0.1 millimeter. The positive y-axis extends upward.
MM_TEXT	Maps one logical unit to one pixel. The positive y-axis extends downward.
MM_TWIPS	Maps one logical unit to 1/1440 inch (1/20 of a point; a point is 1/72 inch). The positive y-axis extends upward.

2.5.1 Constrained Mapping Modes

GDI classifies six of the mapping modes as constrained mapping modes. 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. The relationship of logical units to physical units for each constrained mapping mode follows:

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

Note The MM_HIENGLISH, MM_HIMETRIC, MM_LOENGLISH, MM_LOMETRIC, and MM_TWIPS mapping modes sometimes map logical units to device units in ways that do not correspond exactly to the preceding table. This typically occurs on displays; for example, on an VGA display there is a 33 percent increase in the dimensions of the device units.

The increase in the dimensions of device units occurs so that the same output looks equally crisp and readable whatever the device resolution and the display technology for the device. An application can use the **GetDeviceCaps** function with the LOGPIXELSX and LOGPIXELSY indices to discover the scaling factor.

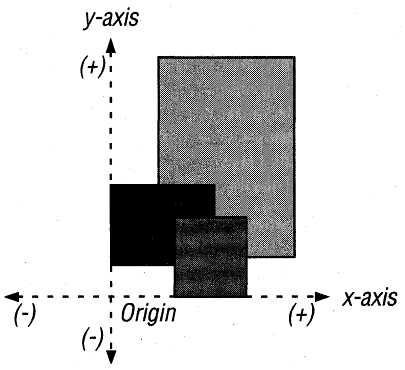
In each of the six constrained modes, one logical unit is mapped to a predefined physical unit. For instance, the MM_TEXT mapping mode maps one logical unit to one device pixel, and the MM_LOENGLISH mapping mode maps one logical unit to 0.01 inch on the device. Examples for these two modes follow.

2.5.1.1 MM_TEXT Mapping Mode

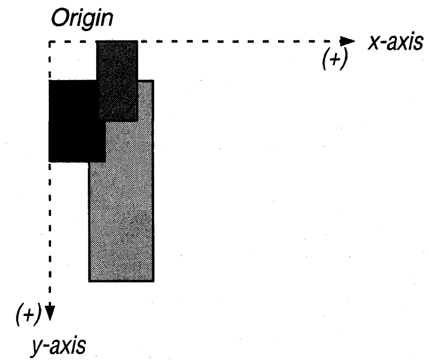
The default mapping mode is MM_TEXT. In this mapping mode, one logical unit is mapped to one pixel on the device or display.

The following illustration shows three rectangles created by a Windows application by using the MM_TEXT mapping mode. The drawing on the left illustrates the logical coordinate space, and the one on the right illustrates the device, or physical, coordinate 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 the positive y-axis extends downward in the physical-coordinate system.

Logical coordinate system



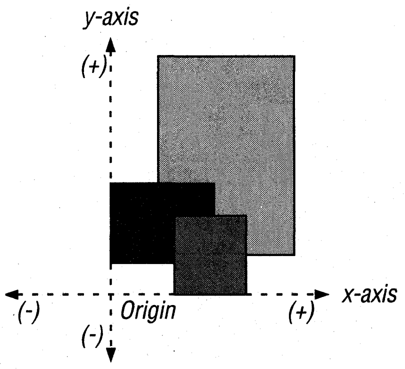
Physical coordinate system



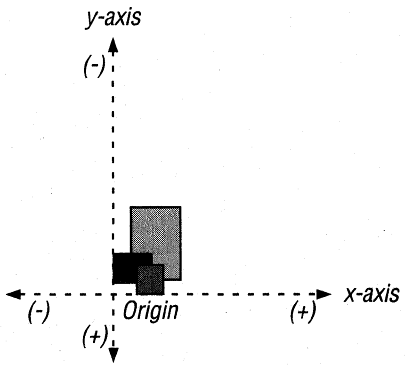
2.5.1.2 MM_LOENGLISH Mapping Mode

The following illustration shows three rectangles created by a Windows application by using the MM_LOENGLISH mapping mode. The drawing on the left illustrates how the rectangles appear in relation to the x-axis and y-axis in the logical coordinate system. The one on the right illustrates how the rectangles appear in relation to the x-axis and y-axis in the physical coordinate system.

Logical coordinate system



Physical coordinate system



2.5.2 Other Mapping Modes

The MM_ISOTROPIC and MM_ANISOTROPIC mapping modes, which are not constrained, 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.

2.5.2.1 Partially Constrained Mapping Mode

An application creates output with equally scaled axes by using the MM_ISOTROPIC mapping mode. As the term isotropic implies, Windows maps 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.

2.5.2.2 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 does not alter either scaling factor in this mode.

2.5.3 Mapping Functions

Mapping functions alter and retrieve information about the GDI mapping modes. Following are the mapping functions:

Function	Description
GetMapMode	Retrieves the current mapping mode.
GetViewportExt	Retrieves the viewport extents of a device context.
GetViewportExtEx	Retrieves viewport extents.
GetViewportOrg	Retrieves the viewport origin of a device context.
GetViewportOrgEx	Retrieves viewport origin.
GetWindowExt	Retrieves the window extents of a device context.
GetWindowExtEx	Retrieves window extents.
GetWindowOrg	Retrieves the window origin of a device context.
GetWindowOrgEx	Retrieves window origin.
OffsetViewportOrg	Modifies a viewport origin.
OffsetViewportOrgEx	Moves viewport origin.
OffsetWindowOrg	Modifies a window origin.
OffsetWindowOrgEx	Moves window origin.
ScaleViewportExt	Modifies the viewport extents.
ScaleViewportExtEx	Scales viewport extents.
ScaleWindowExt	Modifies the window extents.
ScaleWindowExtEx	Scales window extents.
SetMapMode	Sets the mapping mode of a specified device context.
SetViewportExt	Sets the viewport extents for a device context.
SetViewportExtEx	Sets viewport extents.
SetViewportOrg	Sets the viewport origin for a device context.
SetViewportOrgEx	Sets viewport origin.
SetWindowExt	Sets the window extents for a device context.
SetWindowExtEx	Sets window extents.
SetWindowOrg	Sets the window origin for a device context.
SetWindowOrgEx	Sets the window origin.

2.6 Coordinate Functions

Coordinate functions convert client coordinates to screen coordinates (or vice versa). These functions are useful in graphics-intensive applications. Following are the coordinate functions:

Function	Description
DPtoLP	Converts device points (that is, points relative to the window origin) into logical points.
GetCurrentPosition	Retrieves the current position, in logical coordinates.
GetCurrentPositionEx	Retrieves position in logical units.
LPtoDP	Converts logical points into device points.

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:

  $Lx = (Dx - xVO) * xWE/xVE + xWO$
  $Ly = (Dy - yVO) * yWE/yVE + yWO$

Following are descriptions of the variables used in these transformation equations:

Variable	Description
*x*WO	Window origin x-coordinate
*y*WO	Window origin y-coordinate
*x*WE	Window extent x-coordinate
*y*WE	Window extent y-coordinate
*x*VO	Viewport origin x-coordinate
*y*VO	Viewport origin y-coordinate
*x*VE	Viewport extent x-coordinate
*y*VE	Viewport extent y-coordinate
L*x*	Logical-coordinate system x-coordinate
L*y*	Logical-coordinate system y-coordinate
D*x*	Device x-coordinate
D*y*	Device y-coordinate

The following four ratios are scaling factors used to determine the necessary stretching or compressing of logical units: *x*VE/*x*WE, *y*VE/*y*WE, *x*WE/*x*VE, and *y*WE/*y*VE.

The subtraction and addition of viewport and window origins is referred to as the translational component of the equation.

In addition, applications can use the following functions from the USER API to convert coordinates from one system to another:

Function	Description
ChildWindowFromPoint	Determines which, if any, of the child windows belonging to a given parent window contains a specified point.
ClientToScreen	Converts the client coordinates of a given point on the display to screen coordinates.
ScreenToClient	Converts the screen coordinates of a given point on the display to client coordinates.
WindowFromPoint	Retrieves the handle of the window that contains a given point.

For more information about these USER functions, see Chapter 1, "Window Management."

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 graphics output. An application uses these functions in conjunction with the clipping functions to create clipping regions. (For more information about clipping functions, see the next section, "Clipping Functions.") Following are the region functions:

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.

Function	Description
OffsetRgn	Moves the given region.
PaintRgn	Fills the region with the selected brush pattern.
PtInRegion	Tests whether a point is within a region.
RectInRegion	Tests whether any part of a rectangle is within a region.
SetRectRgn	Changes a region into a specified 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 a portion of the client area that is outside the clipping region will not be visible. Clipping regions are useful in Windows applications that need to save one part of the client area and simultaneously send output to another. Following are the clipping functions:

Function	Description
ExcludeClipRect	Excludes a rectangle from the clipping region.
GetBoundsRect	Returns the current accumulated bounding rectangle for the specified device context.
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.
SetBoundsRect	Controls the accumulation of bounding-rectangle information for the specified device context.

2.9 Line Output

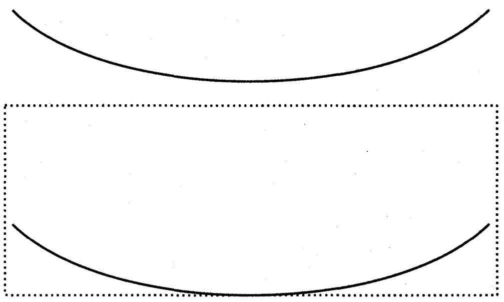
Line output functions require coordinates in logical units, which GDI uses to draw a line in logical space. (The use of logical units ensures device independence in Windows.) GDI maps this line from the logical space to pixels on the device. The number of logical units that GDI maps to a pixel depends on the current mapping mode. When GDI draws a line, it excludes the last specified point.

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 combined (dotted and dashed). Once an application creates a new pen, it can select the pen into a display context by using the **SelectObject** function.

2.9.1 Arcs

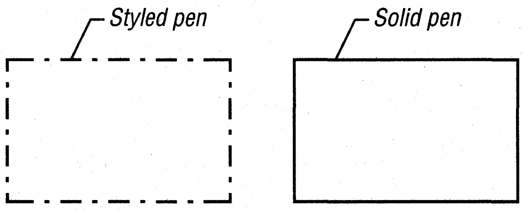
The **Arc** function uses a bounding rectangle to define the size of an arc. The bounding rectangle is hidden; GDI uses it only to describe the location and size of the arc.

The upper portion of the following illustration shows an arc as it would appear on a display. The lower portion shows the arc suspended in the bounding rectangle used by GDI to determine the size and shape of the arc.



2.9.2 Simple Lines

Simple line output can be created by using 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.



2.9.3 Line-Output Functions

Line-output functions create simple and complex line output with the selected pen. Following are the line-output functions:

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.
MoveToEx	Moves the current position.
Polyline	Draws a set of line segments.

2.10 Ellipses and Polygons

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 maps the object from logical space 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.

2.10.1 Rectangles

The **Rectangle** function draws a rectangle, using the current pen. The **RoundRect** function also draws a rectangle, but with rounded rather than square corners.

When GDI draws a rectangle, it uses four arguments. The first two arguments specify the upper-left corner of the rectangle. 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 (x_1, y_1) and the second point is specified by (x_2, y_2), the rectangle's upper-left corner will be (x_1, y_1) and the lower-right corner will be $(x_2 - 1, y_2 - 1)$.

2.10.2 Bounding Rectangles

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

2.10.3 Ellipse and Polygon Functions

Ellipse and polygon functions, which draw ellipses and polygons, are particularly useful in drawing and charting applications. GDI draws the perimeter of each object with the selected pen and fills the interior by using the selected brush. Following are the ellipse and polygon functions:

Function	Description
Chord	Draws a chord.
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.

2.11 Bitmap Functions

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 display surface of the device. Bitmaps are useful in drawing, charting, and word-processing applications because they prepare images in memory and then quickly copy them to the display.

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.

Bitmap functions display bitmaps. Following are the bitmap functions:

Function	Description
BitBlt	Copies a bitmap from a source to a destination device.
CreateBitmap	Creates a bitmap.
CreateBitmapIndirect	Creates a bitmap described in a 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.

Function	Description
FloodFill	Fills the display surface within a border.
GetBitmapBits	Retrieves the bits in memory for a specific bitmap.
GetBitmapDimension	Retrieves the height and width of a bitmap.
GetBitmapDimensionEx	Retrieves the height and width of a bitmap.
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.
SetBitmapDimensionEx	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 (compressing or stretching the bitmap, if necessary).

In addition, applications can use the **LoadBitmap** function from the USER API to load a bitmap from a resource file. For more information about this USER function, see Chapter 1, "Window Management."

2.12 Device-Independent Bitmap Functions

Microsoft Windows provides a set of functions that define and manipulate color bitmaps so that they can be appropriately displayed on a device with a given resolution, regardless of the method used by the device to represent color in memory. These functions translate a device-independent bitmap (DIB) specification into a device-specific format.

A DIB specification consists of two parts:

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

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

Following are the DIB functions:

Function	Description
CreateDIBitmap	Creates a device-specific memory bitmap from a DIB specification and, optionally, initializes bits in the bitmap. This function is similar to the **CreateBitmap** function.
GetDIBits	Retrieves the bits in memory for a specific bitmap in device-independent form. This function is similar to the **GetBitmapBits** function.
SetDIBits	Sets bits of a memory bitmap from a DIB. This function is similar to the **SetBitmapBits** function.
SetDIBitsToDevice	Sets bits on a device surface directly from a DIB.
StretchDIBits	Moves a DIB from a source rectangle into a destination rectangle, stretching or compressing the bitmap as required.

2.13 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. Following are the GDI text functions:

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). It can also be a clipping region.
GetTextAlign	Returns a mask of the text alignment flags.
GetTextCharacterExtra	Retrieves the current setting for the amount of intercharacter spacing.
GetTextExtent	Uses the current font to compute the width and height of text.
GetTextExtentPoint	Retrieves dimensions of string.
SetTextAlign	Positions a string of text on a display or device.
SetTextCharacterExtra	Sets the amount of intercharacter spacing.
SetTextJustification	Justifies a text line.
TextOut	Writes a character string using the current font.

The USER API also includes the following text functions:

Function	Description
DrawText	Draws formatted text into a rectangle.
GetTabbedTextExtent	Computes the width and height of a line of text containing tab characters.
GrayString	Draws gray text by writing the text in a memory bitmap and graying the bitmap. Then it copies the bitmap to the display.
TabbedTextOut	Writes a character string with expanded tabs, using the current font.

For more information about these USER functions, see Chapter 1, "Window Management."

2.14 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. Following are the font functions:

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.
CreateScalableFontResource	Creates a font resource file containing the font directory information and the font module name for a specified scalable font file.
EnumFontFamilies	Enumerates the fonts in a specified font family that are available on a given device. (Supersedes the **EnumFonts** function.)
EnumFonts	Enumerates the fonts available on a given device. Superseded by the **EnumFontFamilies** function.
GetAspectRatioFilter	Retrieves the setting for the current aspect-ratio filter.
GetAspectRatioFilterEx	Retrieves current aspect-ratio filter.

Function	Description
GetCharABCWidths	Retrieves the widths of consecutive characters in a specified range from the current TrueType font.
GetCharWidth	Retrieves the widths of individual characters in a range of consecutive characters from the current font.
GetFontData	Retrieves font metric data from a TrueType font file.
GetGlyphOutline	Retrieves the outline curve or bitmap for an outline character in the current font.
GetOutlineTextMetrics	Fills a buffer with metrics for the selected TrueType font.
GetRasterizerCaps	Retrieves flags indicating whether TrueType fonts are installed in the system.
GetTextFace	Copies the current font name to a buffer.
GetTextMetrics	Fills a buffer with metrics for the selected font.
RemoveFontResource	Removes a font resource from the font table.
SetMapperFlags	Alters the algorithm the font mapper uses.

For information about using font functions in an application, see the *Microsoft Windows Guide to Programming.*

2.15 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.

2.15.1 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** function returns; 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 consists of all GDI functions that create output (rather than functions that provide state information, such as the **GetDeviceCaps** function). The following list shows GDI functions that an application can use in a metafile:

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

To create output in 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 and passing to the function the metafile handle obtained from **CloseMetaFile** and a device-context handle for the device on which the metafile is to be played.

The device context that the **CreateMetaFile** function 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 the **PlayMetaFile** function. Then, when **PlayMetaFile** returns, the application can call the **RestoreDC** function 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. For information about the format of metafile records and descriptions of their contents, see the *Microsoft Windows Programmer's Reference, Volume 4.*

2.15.2 Storing a Metafile

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

To store the metafile in memory, an application calls the **CreateMetaFile** function and passes NULL as the function parameter. The application can free the memory that Windows uses to store the metafile by calling the **DeleteMetaFile** function. This function removes a metafile from memory and invalidates its handle. **DeleteMetaFile** has no effect on disk files.

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

- When the application calls the **CreateMetaFile** function to open a metafile, it passes a filename as the function parameter; the metafile is then 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 name of the disk file to which the metafile will be saved.

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.

2.15.3 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 a callback function supplied by the application and passes it the following information:

- 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 the record, to copy it, or to modify it.

The **PlayMetaFileRecord** function plays a metafile record by executing the GDI function contained in the record.

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 that 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 0. If the second object is a pen, it is given index 1, and so on. For information about the format of the handle table, see the description of the **HANDLETABLE** structure in the *Microsoft Windows Programmer's Reference, Volume 3.*

2.15.4 Metafile Functions

Metafile functions close, copy, create, delete, retrieve, play, and return information about metafiles. Following are the metafile functions:

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 of a metafile.
GetMetaFileBits	Stores a metafile as a collection of bits in a global memory object.
PlayMetaFile	Plays the contents of a specified metafile.
PlayMetaFileRecord	Plays a metafile record.
SetMetaFileBits	Creates a memory metafile.
SetMetaFileBitsBetter	Creates a memory block from a metafile.

2.16 Device-Control Functions

Device-control functions retrieve information about a device and modify its initialization state. Following are the device-control functions:

Function	Description
DeviceCapabilities	Retrieves capabilities of a printer driver.
DeviceMode	Sets the current printing modes for a device by prompting the user with a dialog box.

Function	Description
ExtDeviceMode	Retrieves or modifies device initialization information for a given printer driver or displays a driver-supplied dialog box for configuring the driver.
GetDeviceCaps	Retrieves device-specific information about a given display device.
ResetDC	Updates the specified device context, based on the information in a **DEVMODE** structure.

The printer driver, rather than GDI, provides the **DeviceCapabilities**, **Device-Mode**, and **ExtDeviceMode** functions.

2.17 Printer Functions

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

Following are the eight printer functions in Windows 3.1, which supersede many of the printer escapes:

Function	Description
AbortDoc	Ends the current print job and erases everything drawn since the last call to the **StartDoc** function.
EndDoc	Ends a print job.
EndPage	Informs the printer that the application has finished writing to a page.
QueryAbort	Informs the abort procedure for a printing application that a print job should be stopped.
SetAbortProc	Sets the application-supplied abort procedure that allows a print job to be canceled during spooling.
SpoolFile	Places a file into the spooler queue.
StartDoc	Starts a print job.
StartPage	Prepares the printer driver to begin accepting data.

For information about printing from Windows applications, see the *Microsoft Windows Guide to Programming*.

2.18 Related Topics

For more information about USER API functions, see Chapter 1, "Window Management."

For an introduction to using font functions in an application and to printing from Windows applications, see the *Microsoft Windows Guide to Programming.*

For more information about the **HANDLETABLE** structure and the format of metafile records, see the *Microsoft Windows Programmer's Reference, Volumes 3* and *4*, respectively.

System Services

Chapter 3

This chapter describes the system services interface functions for the Microsoft Windows operating system. 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 Windows initialization file, assist in system debugging, carry out communications through the system's input and output (I/O) ports, create and open files, and create sounds using the system's sound generator.

3.1 Module-Management Functions

Module-management functions alter and retrieve information about Windows modules, which are loadable, executable units of code and data. Following are the module-management functions:

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 handle of a specified 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 memory-management 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 in the data segment of a Windows application. Following are the memory-management functions:

Function	Description
GetFreeSpace	Retrieves the number of bytes available in the global heap.
GetFreeSystemResources	Returns the percentage of free system-resource space.
GetWinFlags	Retrieves information about the system-memory configuration.
GlobalAlloc	Allocates memory from the global heap.
GlobalCompact	Compacts global memory to generate free bytes.
GlobalDosAlloc	Allocates global memory that can be accessed by MS-DOS.
GlobalDosFree	Frees global memory previously allocated by the **GlobalDosAlloc** function.
GlobalFlags	Returns the flags and lock count of a global memory object.
GlobalFree	Removes a global memory object and invalidates the handle of the memory object.
GlobalHandle	Retrieves the handle of a global memory object.
GlobalLock	Retrieves a pointer to a global memory object specified by a handle. Except in the case of nondiscardable objects in protected (standard or 386-enhanced) mode, the object is locked in 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 LRU position.
GlobalNotify	Installs a notification procedure for the current task.
GlobalReAlloc	Reallocates a global memory object.
GlobalSize	Returns the size, in bytes, of a global memory object.
GlobalUnlock	Invalidates the pointer to a global memory object previously retrieved by the **GlobalLock** function. If the object is discardable, **GlobalUnlock** decreases the lock count of the object by one.
GlobalUnWire	Decreases the lock count set by the **GlobalWire** function, and unlocks the memory object 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 assigns to an application.

Function	Description
LocalAlloc	Allocates memory from the local heap.
LocalCompact	Compacts local memory.
LocalFlags	Returns the memory type of a local memory object.
LocalFree	Frees a local memory object from memory if the lock count is zero and invalidates the handle of the memory object.
LocalHandle	Retrieves the handle of a local memory object.
LocalInit	Initializes a local heap in the specified segment.
LocalLock	Locks the local memory object by increasing its lock count.
LocalReAlloc	Reallocates a local memory object.
LocalShrink	Shrinks the local heap.
LocalSize	Returns the size, in bytes, of a local memory object.
LocalUnlock	Unlocks a local memory object.
LockSegment	Locks a specified data segment in memory.
SetSwapAreaSize	Increases the amount of memory that an application reserves for code segments.
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).
UnlockSegment	Unlocks a specified data segment.

3.3 Segment Functions

Segment functions allocate, free, and convert selectors; lock and unlock memory objects referenced by selectors; and retrieve information about segments. Following are the selector functions:

Function	Description
AllocDStoCSAlias	Accepts a data-segment (DS) selector and returns a code-segment (CS) selector that can be used to execute code in a data segment.
AllocSelector	Allocates a new selector.
FreeSelector	Frees a selector originally allocated by the **AllocDStoCSAlias** or **AllocSelector** function.
GetCodeInfo	Retrieves information about a code segment.
GetSelectorBase	Returns the base of a selector.

Function	Description
GetSelectorLimit	Returns the limit of a selector.
GlobalFix	Prevents a global memory object from moving in linear memory.
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 the memory associated with the specified global selector. If the page-lock count reaches zero, the memory can be moved and paged out to disk.
GlobalUnfix	Unlocks a global memory object previously fixed by the **GlobalFix** function.
LockSegment	Locks a segment in memory.
PrestoChangoSelector	Generates a temporary code selector that corresponds to a given data selector or a temporary data selector that corresponds to a given code selector.
SetSelectorBase	Sets the base of a selector.
SetSelectorLimit	Sets the limit of a selector.
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 make it possible for an assembly-language application to perform certain MS-DOS and NetBIOS interrupts without directly coding the interrupt. This ensures compatibility with future Microsoft products. Following are the operating-system interrupt functions:

Function	Description
DOS3Call	Issues an MS-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 being executed. A task is a single Windows application call. Following are the task functions:

Function	Description
Catch	Copies the current execution environment to a buffer.
ExitWindows	Initiates the standard Windows shutdown procedure.
GetCurrentPDB	Returns the current MS-DOS program database (PDB), also known as the program segment prefix (PSP).
GetCurrentTask	Returns the handle of the current task.
GetDOSEnvironment	Retrieves the environment string of the currently running task.
GetNumTasks	Returns the number of tasks currently being executed in the system.
IsTask	Determines whether a task handle is valid.
SetErrorMode	Controls whether Windows handles MS-DOS Function 24h errors or allows the calling application to handle them.
Throw	Restores the execution environment to the specified values.
Yield	Stops 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. Following are the resource-management functions:

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.

Function	Description
LoadMenu	Loads a menu resource.
LoadResource	Loads a resource.
LoadString	Loads a string resource.
LockResource	Retrieves the absolute memory address of a resource.
SetResourceHandler	Sets up a function to load resources.
SizeofResource	Supplies the size, in bytes, of 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 manipulations. Following are the string-manipulation functions:

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 a Windows character string to an OEM character string.
AnsiToOemBuff	Converts a Windows character 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 alphabetic.
IsCharAlphaNumeric	Determines whether a character is alphanumeric.
IsCharLower	Determines whether a character is lowercase.
IsCharUpper	Determines whether a character is uppercase.
IsDBCSLeadByte	Determines whether a character is a double-byte character set (DBCS) lead byte.
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.
lstrcpy	Copies one string to another. Both strings are identified by long pointers.

Function	Description
lstrlen	Determines the length of a string identified by a long pointer.
OemToAnsi	Converts an OEM character string to a Windows character string.
OemToAnsiBuff	Converts an OEM character string in a buffer to a Windows character string.
ToAscii	Translates a virtual-key code to the corresponding Windows 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). Following are the atom-management functions:

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.
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.

The **MAKEINTATOM** macro can also be used to cast an integer for use as a function argument.

3.9 Initialization-File Functions

Initialization-file functions obtain information from and copy information to a Windows or private (application-specific) initialization file. The Windows initialization file (WIN.INI) is a special ASCII file that contains entry-value pairs representing run-time options for applications. Following are the initialization-file functions:

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 from 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 initialization file to record information that affects it alone. This improves the performance of the application and Windows 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 the information affects the Windows environment or other applications and should send the WM_WININICHANGE message to all top-level windows.

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

3.10 Communication Functions

Communication functions carry out communications through the serial and parallel I/O ports of the system. Following are the communication functions:

Function	Description
BuildCommDCB	Fills a device control block with control codes.
ClearCommBreak	Clears the break state from a communications device.
CloseComm	Closes a communications device after transmitting the current buffer.

Function	Description
EnableCommNotification	Enables/disables WM_COMMNOTIFY posting to window.
EscapeCommFunction	Directs a device to carry out an extended function.
FlushComm	Flushes characters from a communications device.
GetCommError	Fills a buffer with the communication status.
GetCommEventMask	Retrieves and then clears an event mask.
GetCommState	Fills a buffer with a device control block.
OpenComm	Opens a communications device.
ReadComm	Reads the bytes from a communications device into a buffer.
SetCommBreak	Sets a break state on a communications device.
SetCommEventMask	Retrieves and then sets an event mask on a communications device.
SetCommState	Sets a communications 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 read next.
WriteComm	Writes the bytes from a buffer to a communications device.

3.11 Utility Macros and Functions

Utility macros and functions return contents of words and bytes, create unsigned long integers and structures, and perform specialized arithmetic. Following are the utility macros and functions:

Function or macro	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** structure.

Function or macro	Description
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 values for red, green, and blue into a palette-relative RGB **COLORREF** value.
RGB	Converts values for red, green, and blue into an explicit RGB **COLORREF** value.

3.12 File Input and Output Functions

File I/O functions create, open, read from, write to, and close files. Following are the file I/O functions:

Function	Description
GetDriveType	Determines whether a disk drive is removable, fixed, or remote.
GetSystemDirectory	Retrieves the path 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 path of the Windows directory.
_hmemcpy	Copies bytes.
_hread	Reads from a file.
_hwrite	Writes to a file.
_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 to a file.
OpenFile	Creates, opens, reopens, or deletes the specified file.
SetHandleCount	Changes the number of file handles available to a task.

3.13 Debugging Functions

Debugging functions help locate programming errors in an application or library. Following are the debugging functions:

Function	Description
DebugBreak	Causes a breakpoint exception to occur in the calling function.
DebugOutput	Sends messages to the debugging terminal.
DirectedYield	Forces execution of a specified task to continue.
FatalAppExit	Displays a message and then exits the application.
FatalExit	Displays the current state of Windows and prompts for instructions on how to proceed.
GetSystemDebugState	Returns system-state information to a debugger.
GetWinDebugInfo	Queries current system-debugging information.
LockInput	Locks input to all tasks except the current one.
OutputDebugString	Sends a debugging message to the debugger if present, or to the AUX device if the debugger is not present.
QuerySendMessage	Determines if a message originated in a task.
SetWinDebugInfo	Sets current system-debugging information.
ValidateCodeSegments	Determines whether any code segments have been altered by random memory overwrites.
ValidateFreeSpaces	Checks free segments in memory for valid contents.

3.14 Optimization-Tool Functions

Optimization-tool functions control how the Microsoft Windows Profiler software development tool interacts with an application being developed. Following are the optimization-tool 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.

3.15 Application-Execution Functions

Application-execution tasks permit one application to execute another program. Following are the application-execution 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 MS-DOS application. The calling application supplies a string containing the name of the executable file to be run and any command parameters, and it also specifies the initial state of the application 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 an MS-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 that specifies a context identifier in the help file or a string containing a keyword in the help file.

3.16 Related Topics

For an introduction to file input and output, libraries, and memory management, see the *Microsoft Windows Guide to Programming*.

For more information about Windows functions and macros, see the *Microsoft Windows Programmer's Reference*, *Volumes 2* and *3*.

For information about debugging and optimization tools, see *Microsoft Windows Programming Tools*.

Extension Libraries

Part 2

Common Dialog Box Library

Chapter 4

Common dialog boxes make it easier for you to develop applications for the Microsoft Windows operating system. A common dialog box is a dialog box that an application displays by calling a single function rather than by creating a dialog box procedure and a resource file containing a dialog box template. The dynamic-link library COMMDLG.DLL provides a default procedure and template for each type of common dialog box. Each default dialog box procedure processes messages and notifications for a common dialog box and its controls. A default dialog box template defines the appearance of a common dialog box and its controls.

In addition to simplifying the development of Windows applications, a common dialog box assists users by providing a standard set of controls for performing certain operations. As Windows developers begin using the common dialog boxes in their applications, users will find that after they master using a common dialog box in one application, they can easily perform the same operations in other applications.

This chapter describes the various common dialog boxes and includes sample code to help you use common dialog boxes in your Windows applications.

Following are the types of common dialog boxes in the order in which they are presented in this chapter:

Name	Description
Color	Displays available colors, from which the user can select one; displays controls that let the user define a custom color.
Font	Displays lists of fonts, point sizes, and colors that correspond to available fonts; after the user selects a font, the dialog box displays sample text rendered with that font.
Open	Displays a list of filenames matching any specified extensions, directories, and drives. By selecting one of the listed filenames, the user indicates which file an application should open.
Save As	Displays a list of filenames matching any specified extensions, directories, and drives. By selecting one of the listed filenames, the user indicates which file an application should save.
Print	Displays information about the installed printer and its configuration. By altering and selecting controls in this dialog box, the user specifies how output should be printed and starts the printing process.
Print Setup	Displays the current list of available printers. The user can select a printer from this list. This common dialog box also provides options for setting the paper orientation, size, and source (when the printer driver supports these options). In addition to being called directly, the Print Setup dialog can be opened from within the Print dialog.
Find	Displays an edit control in which the user can type a string for which the application should search. The user can specify the direction of the search, whether the application should match the case of the specified string, and whether the string to match is an entire word.

Name	Description
Replace	Displays two edit controls in which the user can type strings: the first string identifies a word or value that the application should replace, and the second string identifies the replacement word or value.

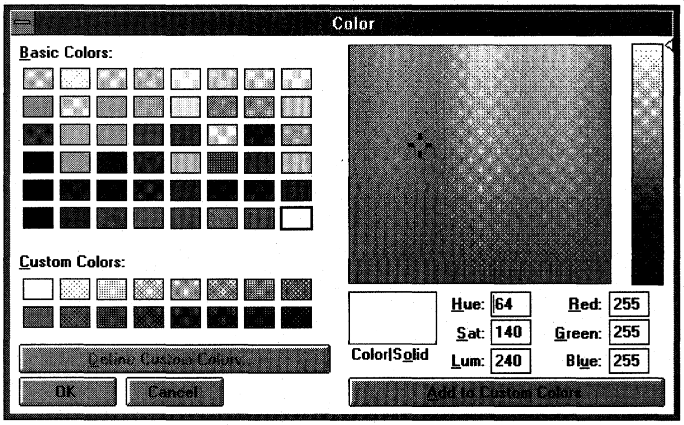
Applications that use the common dialog boxes should specify at least 8K for the stack size, as shown in the following example:

```
NAME cd

EXETYPE WINDOWS

STUB     'WINSTUB.EXE'

CODE     PRELOAD MOVEABLE DISCARDABLE

DATA     PRELOAD MOVEABLE MULTIPLE

HEAPSIZE  1024

STACKSIZE 8192

EXPORTS
  FILEOPENHOOKPROC  @1
```

4.1 Using Color Dialog Boxes

The Color dialog box contains controls that make it possible for a user to select and create colors.

Following is a Color dialog box.



Hue: 64
Sat: 140
Lum: 240
Red: 255
Green: 255
Blue: 255

The Basic Colors control displays up to 48 colors. The actual number of colors displayed is determined by the display driver. For example, a VGA driver displays 48 colors, and a monochrome display driver displays only 16. With the Basic Colors control, the user can select a displayed color.

To display the Custom Colors control, the user clicks the Define Custom Colors button. The Custom Colors control displays custom colors. The user can select one of the 16 rectangles in this control and then create a new color by using one of the following methods:

- Specifying red, green, and blue (RGB) values by using the Red, Green, and Blue edit controls, and then choosing the Add to Custom Colors button to display the new color in the selected rectangle.
- Moving the cursor in the color spectrum control (at the upper-right of the dialog box) to select hue and saturation values; moving the cursor in the luminosity control (the rectangle to the right of the spectrum control); and then choosing the Add to Custom Colors button to display the new color in the selected rectangle.
- Specifying hue, saturation, and luminosity (HSL) values by using the Hue, Sat, and Lum edit controls and then choosing the Add to Custom Colors button to display the new color in the selected rectangle.

The Color|Solid control displays the dithered and solid colors that correspond to the user's selection. (A dithered color is a color created by combining one or more pure or solid colors.) The **Flags** member of the **CHOOSECOLOR** structure contains a flag bit that, when set, displays a Help button. For more information about the **CHOOSECOLOR** structure, see the *Microsoft Windows Programmer's Reference, Volume 3.*

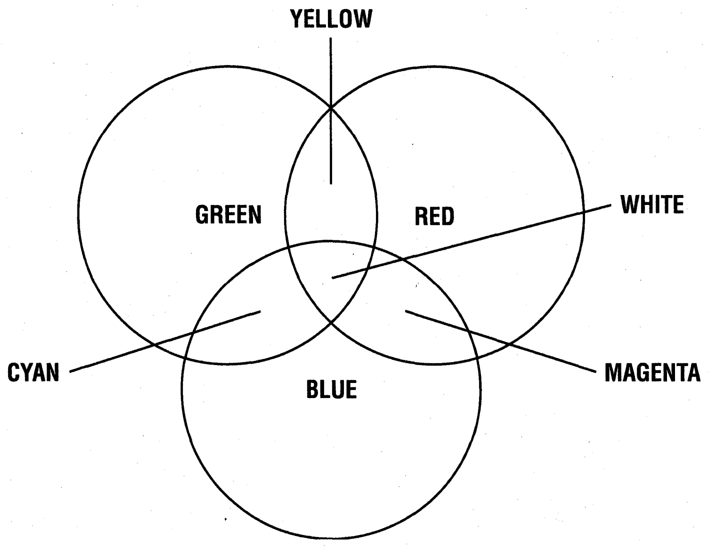
An application can display the Color dialog box in one of two ways: fully open or partially open. When the Color dialog box is displayed partially open, the user cannot change the custom colors.

4.1.1 Color Models Used by the Color Dialog Box

The Color dialog box uses two models for specifying colors: the RGB model and the HSL model. Regardless of the model used, internal storage is accomplished by use of the RGB model.

4.1.1.1 RGB Color Model

The RGB model is used to designate colors for displays and other devices that emit light. Valid red, green, and blue values are in the range 0 through 255, with 0 indicating the minimum intensity and 255 indicating the maximum intensity. The following illustration shows how the primary colors red, green, and blue can be combined to produce four additional colors. (With display devices, the color black results when the red, green, and blue values are set to 0—that is, with display technology, black is the absence of all colors.)



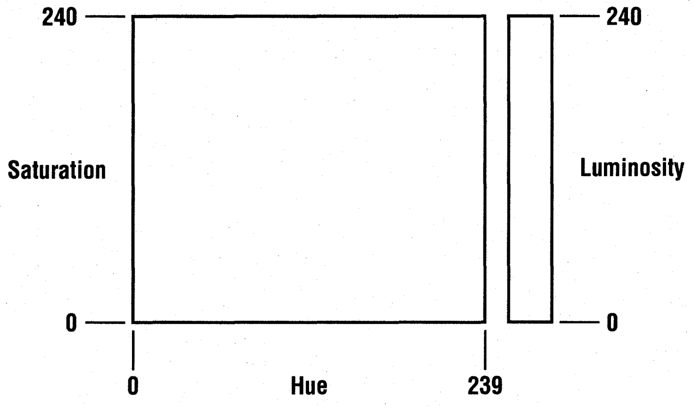
Following are eight colors and their associated RGB values:

Color	RGB values
Red	255, 0, 0
Green	0, 255, 0
Blue	0, 0, 255
Cyan	0, 255, 255
Magenta	255, 0, 255
Yellow	255, 255, 0
White	255, 255, 255
Black	0, 0, 0

Windows stores internal colors as 32-bit RGB values. The high-order byte of the high-order word is reserved; the low-order byte of the high-order word specifies the intensity of the blue component; the high-order byte of the low-order word specifies the intensity of the green component; and the low-order byte of the low-order word specifies the intensity of the red component.

4.1.1.2 HSL Color Model

The Color dialog box provides controls for specifying HSL values. The following illustration shows the color spectrum control and the vertical luminosity control that appear in the Color dialog box and shows the ranges of values the user can specify with these controls.



In the Color dialog box, the saturation and luminosity values must be in the range 0 through 240 and the hue value must be in the range 0 through 239.

4.1.1.3 Converting HSL Values to RGB Values

The dialog box procedure provided in COMMDLG.DLL for the Color dialog box contains code that converts HSL values to the corresponding RGB values. Following are several colors with their associated HSL and RGB values:

Color	HSL values	RGB values
Red	(0, 240, 120)	(255, 0, 0)
Yellow	(40, 240, 120)	(255, 255, 0)
Green	(80, 240, 120)	(0, 255, 0)
Cyan	(120, 240, 120)	(0, 255, 255)
Blue	(160, 240, 120)	(0, 0, 255)
Magenta	(200, 240, 120)	(255, 0, 255)
White	(0, 0, 240)	(255, 255, 255)
Black	(0, 0, 0)	(0, 0, 0)

4.1.2 Using the Color Dialog Box to Display Basic Colors

An application can display the Color dialog box so that a user can select one color from a list of basic screen colors. This section describes how you can provide code and structures in your application that make this possible.

4.1.2.1 Initializing the CHOOSECOLOR Structure

Before you display the Color dialog box you need to initialize a **CHOOSE-COLOR** structure. This structure should be global or declared as a **static** variable. The members of this structure contain information about such items as the following:

- Structure size
- Which window owns the dialog box
- Whether the application is customizing the common dialog box
- The hook function and custom dialog box template to use for a customized version of the Color dialog box
- RGB values for the selected basic color

If your application does not customize the dialog box and you want the user to be able to select a single color from the basic colors, you should initialize the **CHOOSECOLOR** structure in the following manner:

```
/* Color variables */

CHOOSECOLOR cc;
COLORREF clr;
COLORREF aclrCust[16];
int i;

/* Set the custom color controls to white. */

for (i = 0; i < 16; i++)
    aclrCust[i] = RGB(255, 255, 255);

/* Initialize clr to black. */

clr = RGB(0, 0, 0);

/* Set all structure fields to zero. */

memset(&cc, 0, sizeof(CHOOSECOLOR));

/* Initialize the necessary CHOOSECOLOR members. */

cc.lStructSize = sizeof(CHOOSECOLOR);
cc.hwndOwner = hwnd;
cc.rgbResult = clr;
cc.lpCustColors = aclrCust;
cc.Flags = CC_PREVENTFULLOPEN;

if (ChooseColor(&cc))
    .
    . /* Use cc.rgbResult to select the user-requested color. */
    .
```

In the previous example, the array to which the **lpCustColors** member points contains 16 doubleword RGB values that specify the color white, and the CC_PREVENTFULLOPEN flag is set in the **Flags** member to disable the Define Custom Colors button and prevent the user from selecting a custom color.

4.1.2.2 Calling the ChooseColor Function

After you initialize the structure, you should call the **ChooseColor** function. If the function is successful and the user chooses the OK button to close the dialog box, the **rgbResult** member contains the RGB values for the basic color that the user selected.

4.1.3 Using the Color Dialog Box to Display Custom Colors

An application can display the Color dialog box so that the user can create and select a custom color. This section describes how you can provide code and structures in your application that make this possible.

4.1.3.1 Initializing the CHOOSECOLOR Structure

Before you display the Color dialog box, you need to initialize a **CHOOSE-COLOR** structure. This structure should be global or declared as a **static** variable. The members of this structure contain information about such items as the following:

- Structure size
- Which window owns the dialog box
- Whether the application is customizing the common dialog box
- The hook function and custom dialog box template to use for a customized version of the Color dialog box
- RGB values for the custom color control

If your application does not customize the dialog box and you want the user to be able to create and select custom colors, you should initialize the **CHOOSE-COLOR** structure in the following manner:

```
/* Color Variables */

CHOOSECOLOR chsclr;
DWORD dwCustClrs[16] = { RGB(255, 255, 255), RGB(239, 239, 239),
                         RGB(223, 223, 223), RGB(207, 207, 207),
                         RGB(191, 191, 191), RGB(175, 175, 175),
                         RGB(159, 159, 159), RGB(143, 143, 143),
                         RGB(127, 127, 127), RGB(111, 111, 111),
                         RGB(95, 95, 95),    RGB(79, 79, 79),
                         RGB(63, 63, 63),    RGB(47, 47, 47),
                         RGB(31, 31, 31),    RGB(15, 15, 15)
                       };
BOOL fSetColor = FALSE;
int i;

chsclr.lStructSize = sizeof (CHOOSECOLOR);
chsclr.hwndOwner = hwnd;
chsclr.hInstance = NULL;
chsclr.rgbResult = 0L;
chsclr.lpCustColors = (LPDWORD) dwCustClrs;
chsclr.Flags = CC_FULLOPEN;
chsclr.lCustData = 0L;
chsclr.lpfnHook = (FARPROC) NULL;
chsclr.lpTemplateName = (LPSTR)NULL;
```

In the previous example, the array to which **lpCustColors** points contains sixteen 32-bit RGB values that specify 16 scales of gray, and the CC_FULLOPEN flag is set in the **Flags** member to display the complete Color dialog box.

4.1.3.2 Calling the ChooseColor Function

After you initialize the structure, you should call the **ChooseColor** function as shown in the following code fragment:

```
if (fSetColor = ChooseColor(&chsclr))
.
. /* Use chsclr.lpCustColors to select user specified colors*/
.
```

If the function is successful and the user chooses the OK button to close the dialog box, the **lpCustColors** member points to an array that contains the RGB values for the custom colors requested by the application's user.

Applications can exercise more control over custom colors by creating a new message identifier for the string defined by the COLOROKSTRING constant. The application creates the new message identifier by calling the **RegisterWindow-Message** function and passing this constant as the single parameter. After calling **RegisterWindowMessage**, the application receives a message immediately prior to the dismissal of the dialog box. The *lParam* parameter of this message contains a pointer to the **CHOOSECOLOR** structure. The application can use the **lpCust-Colors** member of this structure to check the current color. If the application returns a nonzero value when it processes this message, the dialog box is not dismissed.

Similarly, applications can create a new message identifier for the string defined by the SETRGBSTRING constant. The application's hook function can use the message identifier returned by calling **RegisterWindowMessage** with the SETRGBSTRING constant to set a color in the dialog box. For example, the following line of code sets the color selection to blue:

```
SendMessage(hwhndDlg, wSetRGBMsg, 0, (LPARAM) RGB(0, 0, 255));
```

In this example, wSetRGBMsg is the message identifier returned by the **Register-WindowMessage** function. The *lParam* parameter of the **SendMessage** function is set to the RGB values of the desired color. The *wParam* parameter is not used.

The application can specify any valid RGB values in this call to **SendMessage**. If the RGB values match one of the basic colors, the system selects the basic color and updates the spectrum and luminosity controls. If the RGB values do not match one of the basic colors, the system updates the spectrum and luminosity controls, but the basic color selection remains unchanged.

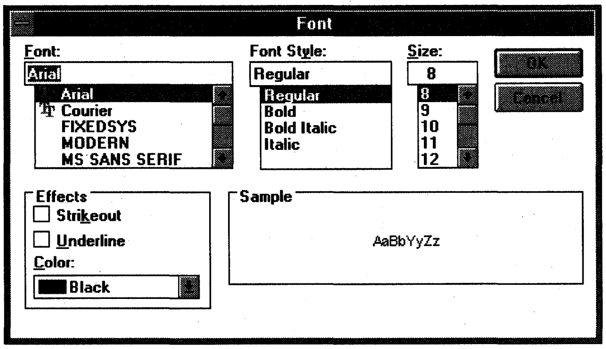
Note that if the Color dialog box is not fully open and the application sends RGB values that do not match one of the basic colors, the system does not update the dialog box. Updates are unnecessary because the spectrum and luminosity controls are not visible when the dialog box is only partially open.

For more information about processing registered window messages, see Section 4.5, "Using Find and Replace Dialog Boxes."

4.2 Using Font Dialog Boxes

The Font dialog box contains controls that make it possible for a user to select a font, a font style (such as bold, italic, or regular), a point size, and an effect (such as underline, strikeout, or a text color).

Following is a Font dialog box.



4.2.1 Displaying the Font Dialog Box in Your Application

The Font dialog box appears after you initialize the members in a **CHOOSEFONT** structure and call the **ChooseFont** function. This structure should be global or declared as a **static** variable. The members of the **CHOOSEFONT** structure contain information about such items as the following:

- The attributes of the font that initially is to appear in the dialog box.
- The attributes of the font that the user selected.
- The point size of the font that the user selected.
- Whether the list of fonts corresponds to a printer, a screen, or both.
- Whether the available fonts listed are TrueType only.
- Whether the Effects box should appear in the dialog box.
- Whether dialog box messages should be processed by an application-supplied hook function.
- Whether the point sizes of the selectable fonts should be limited to a specified range.
- Whether the dialog box should display only what-you-see-is-what-you-get (WYSIWIG) fonts. (These fonts are resident on both the screen and the printer.)
- The color that the **ChooseFont** function should use to render text in the Sample box the first time the application displays the dialog box.
- The color that the user selected for text output.

To display the Font dialog box, an application should perform the following steps:

1. If the application requires printer fonts, retrieve a device-context handle for the printer and use this handle to set the **hDC** member of the **CHOOSEFONT** structure. (If the Font dialog box displays only screen fonts, this member should be set to NULL.)

2. Set the appropriate flags in the **Flags** member of the **CHOOSEFONT** structure. This setting must include CF_SCREENFONTS, CF_PRINTERFONTS, or CF_BOTH.
3. Set the **rgbColors** member of the **CHOOSEFONT** structure if the default color (black) is not appropriate.
4. Set the **nFontType** member of the **CHOOSEFONT** structure using the appropriate constant.
5. Set the **nSizeMin** and **nSizeMax** members of the **CHOOSEFONT** structure if the CF_LIMITSIZE value is specified in the **Flags** member.
6. Call the **ChooseFont** function.

The following example initializes the **CHOOSEFONT** structure and calls the **ChooseFont** function:

```
LOGFONT lf;
CHOOSEFONT cf;

/* Set all structure fields to zero. */

memset(&cf, 0, sizeof(CHOOSEFONT));

cf.lStructSize = sizeof(CHOOSEFONT);
cf.hwndOwner = hwnd;
cf.lpLogFont = &lf;
cf.Flags = CF_SCREENFONTS | CF_EFFECTS;
cf.rgbColors = RGB(0, 255, 255); /* light blue */
cf.nFontType = SCREEN_FONTTYPE;

ChooseFont(&cf);
```

When the user closes the Font dialog box by choosing the OK button, the **ChooseFont** function returns information about the selected font in the **LOGFONT** structure to which the **lpLogFont** member points. An application can use this **LOGFONT** structure to select the font that will be used to render text. The following example selects a font by using the **LOGFONT** structure and renders a string of text:

```
hdc = GetDC(hwnd);
hFont = CreateFontIndirect(cf.lpLogFont);
hFontOld = SelectObject(hdc, hFont);
TextOut(hdc, 50, 150,
    "AaBbCcDdEeFfGgHhIiJjKkLlMmNnOoPpQqRrSsTtUuVvWwXxYyZz", 52);
SelectObject(hdc, hFontOld);
DeleteObject(hFont);
ReleaseDC(hwnd, hdc);
```

An application can also use the WM_CHOOSEFONT_GETLOGFONT message to retrieve the current **LOGFONT** structure for the Font dialog box before the user closes the dialog box.

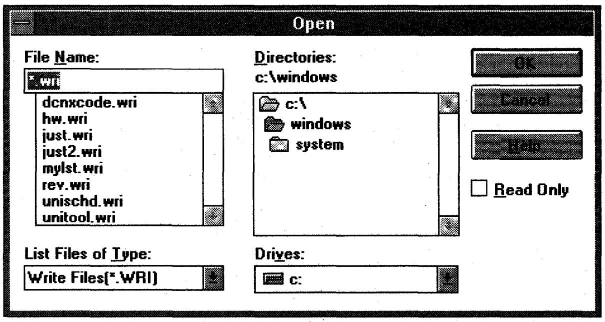
4.3 Using Open and Save As Dialog Boxes

The Open dialog box and the Save As dialog box are similar in appearance. Each contains controls that make it possible for the user to specify the location and name of a file or set of files. In the case of the Open dialog box, the user selects the file or files to be opened; in the case of the Save As dialog box, the user selects the file or files to be saved.

4.3.1 Displaying the Open Dialog Box in Your Application

The Open dialog box appears after you initialize the members of an **OPEN-FILENAME** structure and call the **GetOpenFileName** function.

Following is an Open dialog box.



File Name: *.wri
dcnxcode.wri
hw.wri
just.wri
just2.wri
mylst.wri
rev.wri
unischd.wri
unitool.wri

Directories: c:\windows
c:\
windows
system

List Files of Type: Write Files(*.WRI)

Drives: c:

Before the call to **GetOpenFileName**, structure members contain such data as the name of the directory and the filter that are to appear in the dialog box. (A filter is a filename extension. The common dialog box code uses the extension to filter appropriate filenames from a directory.) After the call, structure members contain such data as the name of the selected file and the number of characters in that filename.

To display an Open dialog box, an application should perform the following steps:

1. Store the valid filters in a character array.
2. Set the **lpstrFilter** member to point to this array.
3. Set the **nFilterIndex** member to the value of the index that identifies the default filter.

4. Set the **lpstrFile** member to point to an array that contains the initial filename and receives the selected filename.
5. Set the **nMaxFile** member to the value that specifies the length of the filename array.
6. Set the **lpstrFileTitle** member to point to a buffer that receives the title of the selected file.
7. Set the **nMaxFileTitle** member to specify the length of the buffer.
8. Set the **lpstrInitialDir** member to point to a string that specifies the initial directory. (If this member does not point to a valid string, it must be set to 0 or point to a string that is set to NULL.)
9. Set the **lpstrTitle** member to point to a string specifying the name that should appear in the title bar of the dialog box. (If this pointer is NULL, the title will be Open.)
10. Initialize the **lpstrDefExt** member to point to the default extension. (This extension can be 0, 1, 2, or 3 characters long.)
11. Call the **GetOpenFileName** function.

The following example initializes an **OPENFILENAME** structure, calls the **GetOpenFileName** function, and opens the file by using the **lpstrFile** member of the structure. The **OPENFILENAME** structure should be global or declared as a **static** variable.

```
OPENFILENAME ofn;
char szDirName[256];
char szFile[256], szFileTitle[256];
UINT  i, cbString;
char  chReplace;    /* string separator for szFilter */
char  szFilter[256];
HFILE hf;

/* Get the system directory name, and store in szDirName. */

GetSystemDirectory(szDirName, sizeof(szDirName));
szFile[0] = '\0';

if ((cbString = LoadString(hinst, IDS_FILTERSTRING,
        szFilter, sizeof(szFilter))) == 0) {
    ErrorHandler();
    return 0L;
}
chReplace = szFilter[cbString - 1]; /* retrieve wildcard */

for (i = 0; szFilter[i] != '\0'; i++) {
    if (szFilter[i] == chReplace)
       szFilter[i] = '\0';
}
```

```
/* Set all structure members to zero. */

memset(&ofn, 0, sizeof(OPENFILENAME));

ofn.lStructSize = sizeof(OPENFILENAME);
ofn.hwndOwner = hwnd;
ofn.lpstrFilter = szFilter;
ofn.nFilterIndex = 1;
ofn.lpstrFile= szFile;
ofn.nMaxFile = sizeof(szFile);
ofn.lpstrFileTitle = szFileTitle;
ofn.nMaxFileTitle = sizeof(szFileTitle);
ofn.lpstrInitialDir = szDirName;
ofn.Flags = OFN_SHOWHELP | OFN_PATHMUSTEXIST | OFN_FILEMUSTEXIST;

if (GetOpenFileName(&ofn)) {
    hf = _lopen(ofn.lpstrFile, OF_READ);
        .
        . /* Perform file operations. */
        .
}
else
    ErrorHandler();
```

The string referred to by the IDS_FILTERSTRING constant in the preceding example is defined as follows in the resource-definition file:

```
STRINGTABLE
BEGIN
 IDS_FILTERSTRING  "Write Files(*.WRI)|*.wri|Word Files(*.DOC)|*.doc|"
END
```

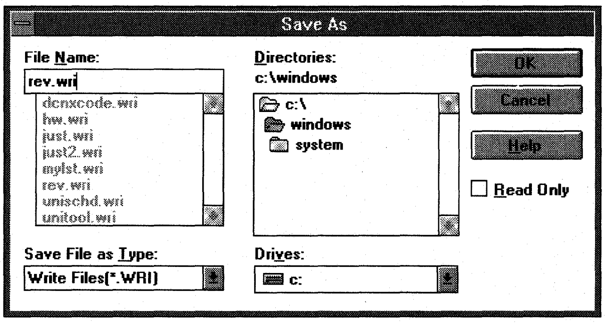
The vertical bars in this string are used as wildcards. After using the **LoadString** function to retrieve the string, the wildcards are replaced with NULL. The wildcard can be any unique character and must be included as the last character in the string. Initializing strings in this manner guarantees that the parts of the string are contiguous in memory and that the string is terminated with two null characters.

Applications that can open files over a network can create a new message identifier for the string defined by the SHAREVISTRING constant. The application creates the new message identifier by calling the **RegisterWindowMessage** function and passing this constant as the single parameter. After calling **RegisterWindowMessage**, the application is notified whenever a sharing violation occurs during a call to the **OpenFile** function. For more information about processing registered window messages, see Section 4.5, "Using Find and Replace Dialog Boxes."

4.3.2 Displaying the Save As Dialog Box in Your Application

The Save As dialog box appears after you initialize the members of an **OPEN-FILENAME** structure and call the **GetSaveFileName** function.

Following is a Save As dialog box.



Before the call to **GetSaveFileName**, structure members contain such data as the name of the initial directory and a filter string. After the call, structure members contain such data as the name of the file to be saved and the number of characters in that filename.

The following example initializes an **OPENFILENAME** structure, calls **GetSave-FileName** function, and saves the file. The **OPENFILENAME** structure should be global or declared as a **static** variable.

```
OPENFILENAME ofn;
char szDirName[256];
char szFile[256], szFileTitle[256];
UINT  i, cbString;
char  chReplace;    /* string separator for szFilter */
char  szFilter[256];
HFILE hf;

/*
 * Retrieve the system directory name, and store it in
 * szDirName.
 */

GetSystemDirectory(szDirName, sizeof(szDirName));

if ((cbString = LoadString(hinst, IDS_FILTERSTRING,
        szFilter, sizeof(szFilter))) == 0) {
    ErrorHandler();
    return 0;
}
```

```
chReplace = szFilter[cbString - 1]; /* retrieve wildcard */

for (i = 0; szFilter[i] != '\0'; i++) {
    if (szFilter[i] == chReplace)
       szFilter[i] = '\0';
}

/* Set all structure members to zero. */

memset(&ofn, 0, sizeof(OPENFILENAME));

/* Initialize the OPENFILENAME members. */

szFile[0] = '\0';

ofn.lStructSize = sizeof(OPENFILENAME);
ofn.hwndOwner = hwnd;
ofn.lpstrFilter = szFilter;
ofn.lpstrFile= szFile;
ofn.nMaxFile = sizeof(szFile);
ofn.lpstrFileTitle = szFileTitle;
ofn.nMaxFileTitle = sizeof(szFileTitle);
ofn.lpstrInitialDir = szDirName;
ofn.Flags = OFN_SHOWHELP | OFN_OVERWRITEPROMPT;

if (GetSaveFileName(&ofn)) {
    .
    . /* Perform file operations. */
    .
}
else
    ErrorHandler();
```

The string referred to by the IDS_FILTERSTRING constant in the preceding example is defined in the resource-definition file. It is used in exactly the same way as the IDS_FILTERSTRING constant discussed in Section 4.3.1, "Displaying the Open Dialog Box in Your Application."

4.3.3 Monitoring List Box Controls in an Open or Save As Dialog Box

An application can monitor list box selections in order to process and display data in custom controls. For example, an application can use a custom control to display the total length, in bytes, of all of the files selected in the File Name box. One method the application can use to obtain this value is to recompute the total count of bytes each time the user selects a file or cancels the selection of a file. A faster method is for the application to use the LBSELCHSTRING message to identify a new selection and add the corresponding file length to the value that appears in the custom control. (Note that in this example, the custom control is a standard

Windows control that you identify in a resource file template for one of the common dialog boxes.)

An application registers the selection-change message with the **RegisterWindowMessage** function. Once the application registers the message, it uses this function's return value to identify messages from the dialog box. The message is processed in the application-supplied hook function for the common dialog box. The *wParam* parameter of each message identifies the list box in which the selection occurred. The low-order word of the *lParam* parameter identifies the list box item. The high-order word of the *lParam* parameter is one of the following values:

Value	Meaning
CD_LBSELCHANGE	Specifies that the item identified by the low-order word of *lParam* was the item in a single-selection list box.
CD_LBSELSUB	Specifies that the item identified by the low-order word of *lParam* is no longer selected in a multiple-selection list box.
CD_LBSELADD	Specifies that the item identified by the low-order word of *lParam* was selected from a multiple-selection list box.
CD_LBSELNOITEMS	Specifies that no items exist in a multiple-selection list box.

For an example that registers a common dialog box message, see Section 4.5, "Using Find and Replace Dialog Boxes."

4.3.4 Monitoring Filenames in an Open or Save As Dialog Box

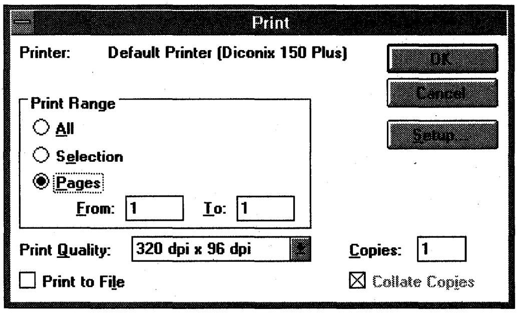
Applications can alter the normal processing of an Open or Save As dialog box by monitoring which filename the user types and by performing other, unique operations. For example, one application could prevent the user from closing the dialog box if the selected filename is prohibited; another application could make it possible for the user to select multiple filenames.

To monitor filenames, an application should register the **FILEOKSTRING** message. An application registers this message by calling the **RegisterWindowMessage** function and passing the message name as its single parameter. After the message is registered, the dialog box procedure in COMMDLG.DLL uses it to signal that the user has selected a filename and chosen the OK button and that the dialog box has checked the filename and is ready to return. The dialog box procedure signals these actions by sending the message to the application's hook function. After receiving the message, the hook function should return a value to the dialog box procedure that called it. If the hook function did not process the message, it should return 0; if the hook function did process the message and the dialog box should close, the hook function should return 0; if the hook function did process the message but the dialog box should not close, the hook function should return 1. (All other return values are reserved.)

4.4 Using Print and Print Setup Dialog Boxes

A Print dialog box contains controls that let a user configure a printer for a particular print job. The user can make such selections as print quality, page range, and number of copies (if the printer supports multiple copies).

Following is a Print dialog box.

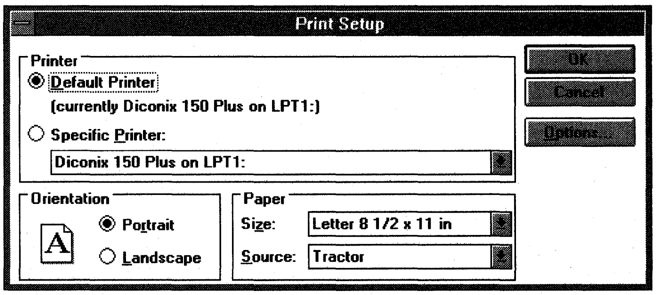


Printer: Default Printer (Diconix 150 Plus)

Print Range
○ All
○ Selection
● Pages
From: 1 To: 1

Print Quality: 320 dpi x 96 dpi
Copies: 1
☐ Print to File
☒ Collate Copies

Choosing the Setup button in the Print dialog box displays the following Print Setup dialog box for a PostScript printer.



Printer
● Default Printer
(currently Diconix 150 Plus on LPT1:)
○ Specific Printer:
Diconix 150 Plus on LPT1:

Orientation
● Portrait
○ Landscape

Paper
Size: Letter 8 1/2 x 11 in
Source: Tractor

The Print Setup dialog box provides controls that make it possible for the user to reconfigure the selected printer.

4.4.1 Device Drivers and the Print Dialog Box

The Print dialog box differs from other common dialog boxes in that part of its dialog box procedure resides in COMMDLG.DLL and part in a printer driver. A printer driver is a program that configures a printer, converts graphics device interface (GDI) commands to low-level printer commands, and stores commands for a particular print job in a printer's queue.

A printer driver exports a function called **ExtDeviceMode**, which displays a dialog box and its controls. In previous versions of Windows, an application called the **LoadLibrary** function to load a device driver and the **GetProcAddress** function to obtain the address of the **ExtDeviceMode** function. This is no longer necessary with the Windows common dialog box interface. Instead of calling **LoadLibrary** and **GetProcAddress**, a Windows application can call a single function, **PrintDlg**, to display the Print dialog box and begin a print job. The code for **PrintDlg** resides in COMMDLG.DLL. The dialog box that appears when an application calls **PrintDlg** differs slightly from the dialog box that appears when the application calls directly into the device driver. The functionality is very similar in spite of the different appearance.

4.4.2 Displaying a Print Dialog Box for the Default Printer

To display a Print dialog box for the default printer, an application must initialize a **PRINTDLG** structure and then call the **PrintDlg** function.

The members of the **PRINTDLG** structure can contain information about such items as the following:

- The printer device context
- Values that should appear in the dialog box controls
- The hook function and custom dialog box template to use for a customized version of the Print dialog box or Print Setup dialog box

An application can display a Print dialog box for the currently installed printer by performing the following steps:

1. Setting the PD_RETURNDC flag in the **Flags** member of the **PRINTDLG** structure. (This flag should only be set if the application requires a device-context handle.)
2. Initializing the **lStructSize**, **hDevMode**, and **hDevNames** members.
3. Calling the **PrintDlg** function and passing a pointer to the **PRINTDLG** structure just initialized.

Setting the PD_RETURNDC flag causes **PrintDlg** to display the Print dialog box and return a handle identifying a printer device context in the **hDC** member of the **PRINTDLG** structure. (The application passes the device-context handle as the first parameter to the GDI functions that render output on the printer.)

The following example initializes the members of the **PRINTDLG** structure and calls the **PrintDlg** function prior to printing output. This structure should be global or declared as a **static** variable.

```
PRINTDLG pd;

/* Set all structure members to zero. */

memset(&pd, 0, sizeof(PRINTDLG));

/* Initialize the necessary PRINTDLG structure members. */

pd.lStructSize = sizeof(PRINTDLG);
pd.hwndOwner = hwnd;
pd.Flags = PD_RETURNDC;

/* Print a test page if successful. */

if (PrintDlg(&pd) != 0) {
    Escape(pd.hDC, STARTDOC, 8, "Test-Doc", NULL);

    /* Print text and rectangle. */

    TextOut(pd.hDC, 50, 50, "Common Dialog Test Page", 23);
    Rectangle(pd.hDC, 50, 90, 625, 105);
    Escape(pd.hDC, NEWFRAME, 0, NULL, NULL);
    Escape(pd.hDC, ENDDOC, 0, NULL, NULL);
    DeleteDC(pd.hDC);
    if (pd.hDevMode != NULL)
       GlobalFree(pd.hDevMode);
    if (pd.hDevNames != NULL)
       GlobalFree(pd.hDevNames);
}
else
    ErrorHandler();
```

4.5 Using Find and Replace Dialog Boxes

The Find dialog box and the Replace dialog box are similar in appearance. You can use the Find dialog box to add string-search capabilities to your application and use the Replace dialog box to add both string-search and string-substitution capabilities.

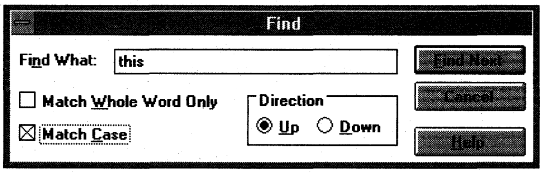
4.5.1 Displaying the Find Dialog Box

The Find dialog box contains controls that make it possible for a user to specify the following:

- The string that the application should find
- Whether the string specifies a complete word or part of a word
- Whether the application should match the case of the specified string

- The direction in which the application should search (preceding or following the current cursor location)
- Whether the application should resume the search, searching for the next occurrence of the string

Following is a Find dialog box.



Find What: this

☐ Match Whole Word Only

☒ Match Case

Direction: [✓ Up] Down

To display the Find dialog box, you need to initialize a **FINDREPLACE** structure and call the **FindText** function. Members of the **FINDREPLACE** structure contain information about such items as the following:

- Which window owns the dialog box
- How the application should perform the search
- A character buffer that is to receive the string

To initialize the **FINDREPLACE** structure, you need to perform the following tasks:

1. Set the **lStructSize** member by using the **sizeof** operator.
2. Set the **hwndOwner** member by using the handle that identifies the owner window of the dialog box.
3. If you are customizing the Find dialog box, set the **hInstance** member to identify the instance of the module that contains your custom dialog box template.
4. Set the **Flags** member to indicate the selection state of the dialog box options. (For example, setting the FR_NOUPDOWN flag disables the Up and Down buttons, setting the FR_NOWHOLEWORD flag disables the Match Whole Word Only check box, and setting the FR_NOMATCHCASE flag disables the Match Case check box).
5. If you are supplying a custom dialog box template or hook function, set additional flags in the **Flags** member.
6. Set the **lpstrFindWhat** member to point to the buffer that will receive the string to be found.
7. Set the **wFindWhatLen** member to specify the size, in bytes, of the buffer to which **lpstrFindWhat** points.
8. Set the **lCustData** member with any custom data your application may need to access.

9. If your application customizes the Find dialog box, set the **lpfnHook** member to point to your hook function.
10. If your application uses a custom dialog box template, set the **lpTemplate-Name** member to point to the string that identifies the template.

The following example initializes the **FINDREPLACE** structure and then calls the **FindText** function. This structure should be global or declared as a **static** variable.

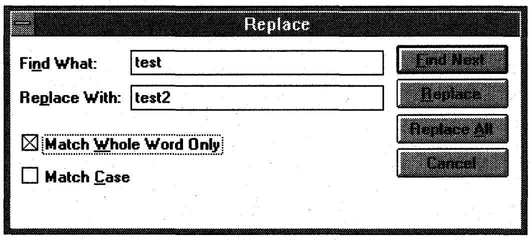
```
FINDREPLACE fr;

/* Set all structure fields to zero. */

memset(&fr, 0, sizeof(FINDREPLACE));

fr.lStructSize = sizeof(FINDREPLACE);
fr.hwndOwner = hwnd;
fr.lpstrFindWhat = szFindWhat;
fr.wFindWhatLen = sizeof(szFindWhat);

hDlg = FindText(&fr);

break;
```

4.5.2 Displaying the Replace Dialog Box

The Replace dialog box is similar to the Find dialog box. However, the Replace dialog box has no Direction box and has three additional controls that make it possible for the user to specify the following:

- The replacement string
- Whether the application should replace the occurrence of the string that is currently highlighted
- Whether the application should replace all occurrences of the string

Following is a Replace dialog box.



Find What: test
Replace With: test2
☒ Match Whole Word Only
☐ Match Case

To display the Replace dialog box, you need to initialize a **FINDREPLACE** structure and call the **ReplaceText** function.

4.5.3 Processing Dialog Box Messages for a Find or Replace Dialog Box

The Find and Replace dialog boxes differ from the other common dialogs in two respects: First, they are modeless; and second, their respective dialog box procedures send messages to the application that calls the **FindText** or **ReplaceText** function. These messages contain data specified by the user in the dialog box controls, such as the direction in which the application should search for a string, whether the application should match the case of the specified string, and whether the application should match the entire string.

To process messages from a Find or Replace dialog box, an application must register the dialog box's unique message, **FINDMSGSTRING**.

The application registers this message with the **RegisterWindowMessage** function. Once the application registers the message, it uses the function's return value to identify messages from the Find or Replace dialog box. The following example registers the message with the **RegisterWindowMessage** function:

```
UINT uFindReplaceMsg;

/* Register the FindReplace message. */

uFindReplaceMsg = RegisterWindowMessage(FINDMSGSTRING);
```

After the application registers this message, it can process messages for the Find or Replace dialog box by using the **RegisterWindowMessage** return value. The following example processes messages for the Find dialog box and then calls its own SearchFile function to locate the string of text. If the user is closing the dialog box (that is, if the **Flags** member of **FINDREPLACE** is FR_DIALOGTERM), the handle should be invalidated and the procedure should return zero.

```
LRESULT CALLBACK MainWndProc(HWND hwnd, UINT msg, WPARAM wParam,
    LPARAM lParam)
{
    FINDREPLACE FAR* lpfr;

    if (msg == uFindReplaceMsg) {
        lpfr = (FINDREPLACE FAR*) lParam;
        SearchFile((BOOL) (lpfr->Flags & FR_DOWN),
            (BOOL) (lpfr->Flags & FR_MATCHCASE));
        return 0;
    }
```

4.6 Customizing Common Dialog Boxes

A custom common dialog box is a common dialog box that has been altered to suit a particular Windows application. The customization may be complex and include the hiding of original controls, the addition of new controls, or a change in the size of the original dialog box; or it may be simple, such as the alteration of a single existing control.

Developers who need to customize a common dialog box must provide a special hook function and, in most cases, a custom dialog box template. Customizations of this kind require a significant amount of additional code—displaying a customized common dialog box is not as simple as initializing the members of a structure and calling a single function.

Applications that subclass controls in any of the common dialog boxes must do so while processing the WM_INITDIALOG message in the application's hook function. This allows the application to receive the control-specific messages first, because it will have subclassed the control after the common dialog box has installed its subclassing procedures. (The previous hook function should be called for all messages that are not handled by the application's subclass function, as is standard for subclassing.)

An application cannot subclass a control by defining a local class to override a specific control type. The reason is that the data segment would not be correctly initialized when the class was called—the data segment would be the common dialog box's data segment, not the application's data segment.

4.6.1 Appropriate and Inappropriate Customizations

From the user's perspective, the chief benefit of the common dialog box is its consistent appearance and functionality from application to application. Therefore, it becomes important that a developer only customize a common dialog box when it is absolutely necessary for an application. Otherwise, the consistent appearance and simple coding interface are lost. Appropriate customizations leave intact as many of the original controls as possible. Increasing the size of the dialog box or adding new controls in available space that already appears in the dialog box would be an appropriate customization. Hiding original controls or otherwise changing the intended functionality of the original controls would be an inappropriate customization.

4.6.2 Hook Functions and Custom Dialog Box Templates

Each common dialog box uses the dialog box procedure and dialog box template provided for it in COMMDLG.DLL. The dialog box procedure processes messages and notifications for the common dialog box and its controls. The dialog box template defines the appearance of the dialog box—its dimensions, its location, and the dimensions and locations of controls that appear within it.

In addition to the provided dialog box procedure and dialog box template, a custom dialog box requires a hook function that you provide and, usually, a custom version of the dialog box template.

4.6.2.1 The Hook Function

The dialog box procedure provided in COMMDLG.DLL for a common dialog box calls the application's hook function if the application sets the appropriate flag and pointer in the structure for that common dialog box. The structure for each common dialog box contains a **Flags** member that specifies whether the application supplies a hook function and contains an **lpfnHook** member that points to the hook function if one exists. If the application sets the **Flags** member to indicate that a hook function exists, it must also set the **lpfnHook** member. The following example sets the **Flags** and **lpfnHook** members of an **OPENFILENAME** structure to support an application's hook function:

```
#define STRICT

#include <windows.h>    /* required for all Windows applications */
#include <commdlg.h>
#include <string.h>
#include "header.h"     /* specific to this program              */

OPENFILENAME ofn;

    /* Get the system directory name, and store in szDirName. */

    GetSystemDirectory((LPSTR)szDirName, 255);

    /* Initialize the OPENFILENAME members. */

    szFile[0] = '\0';
    ofn.lStructSize = sizeof(OPENFILENAME);
    ofn.hwndOwner = hwnd;
    ofn.hInstance = hInst;
    ofn.lpstrFilter = szFilter[0];
    ofn.lpstrCustomFilter = NULL;
    ofn.nMaxCustFilter = 0L;
    ofn.nFilterIndex = 1L;
    ofn.lpstrFile= szFile;
    ofn.nMaxFile = sizeof(szFile);
    ofn.lpstrFileTitle = szFileTitle;
    ofn.nMaxFileTitle = sizeof(szFileTitle);
    ofn.lpstrInitialDir = szDirName;
    ofn.lpstrTitle = NULL;
    ofn.Flags = OFN_ENABLEHOOK | OFN_ENABLETEMPLATE;
    ofn.nFileOffset = 0;
    ofn.nFileExtension = 0;
    ofn.lpstrDefExt = NULL;
    ofn.lpfnHook = MakeProcInstance((FARPROC) FileOpenHookProc, hInst);
    ofn.lpTemplateName = "FileOpen";
```

In the previous example, the **MakeProcInstance** function is called to create a procedure-instance address for the hook function. This address is assigned to the **lpfnHook** member of the **OPENFILENAME** structure. If the hook function is part of a dynamic-link library (rather than an application), the procedure address is obtained by calling the **GetProcAddress** function (instead of **MakeProcInstance**).

The hook function processes any messages or notifications that the custom dialog box requires. With the exception of one message (WM_INITDIALOG), the hook function receives messages and notifications before the dialog box procedure provided in COMMDLG.DLL receives them. In the case of WM_INITDIALOG, the hook function receives the message after the dialog box procedure and should process it as described in the *Microsoft Windows Programmer's Reference, Volume 3*. When the hook function finishes processing a message, it returns a value that indicates whether the dialog box procedure provided in COMMDLG.DLL should also process the message. If the dialog box procedure should process the message, the return value is FALSE; if the dialog box procedure should ignore the message, the return value is TRUE.

To process the message from the OK button after the dialog box procedure processes it, an application must post a message to itself when the OK message is received. When the application receives the message it has posted, the common dialog box procedure will have finished processing messages for the dialog box. This technique is particularly useful when working with the Find and Replace dialog boxes, because the **Flags** member of the **FINDREPLACE** structure does not reflect changes to the dialog box until after the messages have been processed by COMMDLG.DLL.

The following example shows a hook function for a custom Open dialog box:

```
UINT CALLBACK FileOpenHookProc(HWND hdlg, UINT msg, WPARAM wParam,
    LPARAM lParam)
{
    switch(msg) {
        case WM_INITDIALOG:
            return TRUE;
        case WM_COMMAND:

            /* Use IsDlgButtonChecked to set lCustData. */

            if (wParam == IDOK) {

                /* Set backup flag. */

                ofn.lCustData =
                    (DWORD) IsDlgButtonChecked(hdlg, ID_CUSTCHX);
            }

            return FALSE; /* Allow standard processing. */
    }

    /* Allow standard processing. */

    return FALSE;
}
```

This hook function tests a custom check box when the user chooses the OK button. If the check box was selected, the hook function sets the **lCustData** member of the **OPENFILENAME** structure to 1; otherwise, it sets the **lCustData** member to 0.

A hook function should never call the **EndDialog** function. Instead, if a hook function contains code that abnormally terminates a common dialog box, this code should pass the IDABORT value to the dialog box procedure by using the **PostMessage** function as shown in the following example:

```
PostMessage(hDlg, WM_COMMAND, IDABORT, (LONG) FALSE);
```

When a hook function posts the IDABORT value, the common dialog box function returns the value contained in the low word of the *lParam* parameter. For example, if the hook function for **GetOpenFileName** called the **PostMessage** function with (LONG) 100 as the last parameter, **GetOpenFileName** would return 100.

A hook function must be exported in an application's module-definition (.DEF) file as shown in the following example:

```
NAME cd

EXETYPE WINDOWS

STUB    'WINSTUB.EXE'

CODE    PRELOAD MOVEABLE DISCARDABLE
```

```
DATA    PRELOAD MOVEABLE MULTIPLE

HEAPSIZE  1024

STACKSIZE 8192

EXPORTS
  FILEOPENHOOKPROC  @1
```

4.6.2.2 Customizing a Dialog Box Template

The dialog box template provided in COMMDLG.DLL for each common dialog box contains the data that the dialog box procedure uses to display that common dialog box. Most applications that customize a common dialog box also need to create a custom dialog box template to use instead of the dialog box template in COMMDLG.DLL. (A custom dialog box template is not required for all custom dialog boxes. For instance, a template would not be necessary if an application changed a dialog box in a relatively minor way and only in an unusual situation.)

A developer should create a custom dialog box template by modifying the appropriate dialog box template in COMMDLG.DLL. Following are the template filenames and the names of their corresponding common dialog boxes:

Template filename	Corresponding dialog box
COLOR.DLG	Color
FILEOPEN.DLG	Open (single selection)
FILEOPEN.DLG	Open (multiple selection)
FINDTEXT.DLG	Find
FINDTEXT.DLG	Replace
FONT.DLG	Font
PRNSETUP.DLG	Print
PRNSETUP.DLG	Print Setup

The following excerpt is from a custom dialog box template created for an Open dialog box:

```
    CONTROL "&Backup File", ID_CUSTCHX, "button",
            BS_AUTOCHECKBOX | WS_CHILD | WS_TABSTOP | WS_GROUP,
            208,  86,  50,  12

END
```

This entry supports the addition of a new Backup File check box immediately below the existing Read Only check box.

The custom template should be added to the application's resource file.

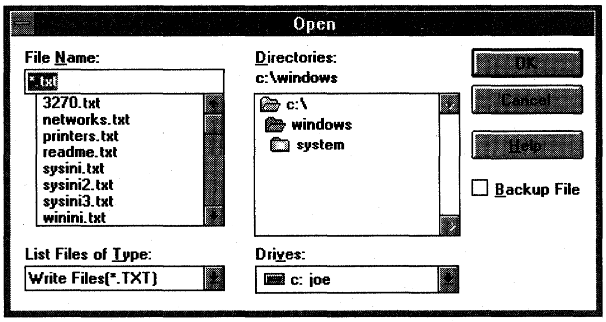
4.6.3 Displaying the Custom Dialog Box

After your application creates the hook function and the dialog box template, it should set the members of the structure for the common dialog box being customized and call the appropriate function to display the custom dialog box.

The following example calls the **GetOpenFileName** function and creates a backup file if the user selected the custom Backup File check box in the custom Open dialog box:

```
/* Open the file and create a backup. */

if (GetOpenFileName(&ofn)) {

    hf = _lopen(ofn.lpstrFile, OF_READWRITE);

    /* Create the backup file. */

    if (ofn.lCustData) {

        /* Process files with extension. */

        if (ofn.nFileExtension){

           for (i=0; i<(int)ofn.nFileExtension; i++)
                szChar[i] = *ofn.lpstrFile++;

        }/*endif */

        /* Process files without extension. */

        else {

           i=0;

           while (*ofn.lpstrFile!='\0')
               szChar[i++] = *ofn.lpstrFile++;

           szChar[i]='.';
        }/*end else*/

        pszNewPAFN = lstrcat(szChar, "BAK");

        /* Create the backup file. */

        hfBackup = _lcreat(pszNewPAFN, 0);
        /* Copy contents of original file to the backup file. */

        while ((cBufLngth=_lread(hf, cBuf1, 256)) == 256)
             _lwrite(hfBackup, cBuf1, cBufLngth);
        _lwrite(hfBackup, cBuf1, cBufLngth);
        _lclose(hfBackup);
    } /*endif GetOpenFileName*/

    /* File operations begin here. */

}   /* endif (GetOpenFileName)    */
```

The following is the custom Open dialog box. The new Backup File check box appears in the lower-right corner.



File Name: *.txt
3270.txt
networks.txt
printers.txt
readme.txt
sysini.txt
sysini2.txt
sysini3.txt
winini.txt

Directories: c:\windows
c:\
windows
system

☐ Backup File

List Files of Type: Write Files(*.TXT)

Drives: c: joe

4.7 Supporting Help for the Common Dialog Boxes

An application can display a Help button in any of the common dialog boxes by setting the appropriate flag in the **Flags** member of the structure for that common dialog box. Following are the structures for the common dialog boxes and the Help flag that corresponds to each structure:

Structure	Flag value
OPENFILENAME	OFN_SHOWHELP
CHOOSECOLOR	CC_SHOWHELP
FINDREPLACE	FR_SHOWHELP
CHOOSEFONT	CF_SHOWHELP
PRINTDLG	PD_SHOWHELP

If an application displays the Help button, it must process the user's request for Help. This can be done either in one of the application's window procedures or in a hook function.

If the application processes the request for Help in one of the application's window procedures, it must first create a new message identifier for the string defined by the HELPMSGSTRING constant. The application creates the new message identifier by calling the **RegisterWindowMessage** function and passing this constant as the single parameter. (For more information about processing registered window messages, see Section 4.5, "Using Find and Replace Dialog Boxes.") In addition to creating a new message identifier, the application must set the **hwndOwner** member of the appropriate structure for the common dialog box so that this member contains the handle of the dialog box's owner window. After the message identifier is created and the **hwndOwner** member is set, the dialog box procedure notifies the window procedure of the owner window whenever the user chooses the Help button.

The following example processes a user's request for Help in the window procedure of its owner window. The **if** statement should be in the **default:** section of the switch statement that processes messages.

```
MyHelpMsg = RegisterWindowMessage(HELPMSGSTRING);
          .
          .
          .
if (message == MyHelpMsg)
   WinHelp(hWnd, "appfile.hlp", HELP_CONTEXT, ID_MY_CONTEXT);
```

If the application processes the request for Help in a hook function, it should test for the following condition in the WM_COMMAND message:

```
wParam == pshHelp
```

When this condition is true, the hook function should call the **WinHelp** function as shown in the preceding example. (To process Help in a hook function, you must include the header file DLGS.H in the source file that contains the hook-function code.)

4.8 Error Detection

Whenever a common dialog box function fails, an application can call the **CommDlgExtendedError** function to find out the cause of the failure. The **CommDlgExtendedError** function returns an error value that identifies the cause of the most recent error.

Six constants are defined in the CDERR.H header file that identify the ranges of error values for categories of errors returned by **CommDlgExtendedError**. Following are these constants in ascending order by value range:

Constant	Meaning
CDERR_GENERALCODES	General error codes for common dialog boxes. These errors are in the range 0x0000 through 0x0FFF.
PDERR_PRINTERCODES	Error codes for the Print common dialog box. These errors are in the range 0x1000 through 0x1FFF.
CFERR_CHOOSEFONTCODES	Error codes for the Font common dialog box. These errors are in the range 0x2000 through 0x2FFF.
FNERR_FILENAMECODES	Error codes for the Open and Save As common dialog boxes. These errors are in the range 0x3000 through 0x3FFF.
FRERR_FINDREPLACECODES	Error codes for the Find and Replace common dialog boxes. These errors are in the range 0x4000 through 0x4FFF.
CCERR_CHOOSECOLORCODES	Error codes for the Color common dialog box. These errors are in the range 0x5000 through 0x5FFF.

4.9 Related Topics

For more information about functions for common dialog boxes, see the *Microsoft Windows Programmer's Reference, Volume 2.*

For more information about common dialog box structures and messages, see the *Microsoft Windows Programmer's Reference, Volume 3.*

Dynamic Data Exchange Management Library

Chapter 5

This chapter describes how to use the Dynamic Data Exchange Management Library (DDEML). The DDEML is a dynamic-link library (DLL) that applications running with the Microsoft Windows operating system can use to share data.

The following topics are related to the information in this chapter:

- Atoms
- Memory management
- Clipboard
- Dynamic-link libraries
- Object linking and embedding (OLE)

Dynamic data exchange (DDE) is a form of interprocess communication that uses shared memory to exchange data between applications. Applications can use DDE for one-time data transfers and for ongoing exchanges in which the applications send updates to one another as new data becomes available.

Dynamic data exchange differs from the clipboard data-transfer mechanism that is also part of the Windows operating system. One difference is that the clipboard is almost always used as a one-time response to a specific action by the user—such as choosing the Paste command from a menu. Although DDE may also be initiated by a user, it typically continues without the user's further involvement.

The DDEML provides an application programming interface (API) that simplifies the task of adding DDE capability to a Windows application. Instead of sending, posting, and processing DDE messages directly, an application uses the functions provided by the DDEML to manage DDE conversations. (A DDE conversation is the interaction between client and server applications.) The DDEML also provides a facility for managing the strings and data that are shared among DDE applications. Instead of using atoms and pointers to shared memory objects, DDE applications create and exchange string handles, which identify strings, and data handles, which identify global memory objects. DDEML provides a service that makes it possible for a server application to register the service names that it supports. The names are broadcast to other applications in the system, which can then use the names to connect to the server. The DDEML also ensures compatibility among DDE applications by forcing them to implement the DDE protocol in a consistent manner.

Existing applications that use the message-based DDE protocol are fully compatible with those that use the DDEML. That is, an application that uses message-based DDE can establish conversations and perform transactions with applications that use the DDEML. Because of the many advantages of the DDEML, new applications should use it rather than the DDE messages.

The DDEML can run on systems that have Microsoft Windows version 3.0 or later installed. The DDEML does not support real mode. To use the API elements of the DDE management library, you must include the DDEML.H header file in your source files, link with DDEML.LIB, and ensure that DDEML.DLL resides in the system's path.

5.1 Basic Concepts

The concepts in this section are key to understanding DDE and the DDEML.

5.1.1 Client and Server Interaction

Dynamic data exchange always takes place between a client application and a server application. The client initiates the exchange by establishing a conversation with the server so that it can send transactions to the server. (A transaction is a request for data or services.) The server responds to these transactions by providing data or services to the client. A server can have many clients at the same time, and a client can request data from multiple servers. Also, an application can be both a client and a server. A client terminates a conversation when it no longer needs a server's data or services.

For example, a graphics application might contain a bar graph that represents a corporation's quarterly profits, and the data for the bar graph might be contained in a spreadsheet application. To obtain the latest profit figures, the graphics application (the client) establishes a conversation with the spreadsheet application (the server). The graphics application then sends a transaction to the spreadsheet application, requesting the latest profit figures.

5.1.2 Transactions and the DDE Callback Function

The DDEML notifies an application of DDE activity that affects the application by sending transactions to the application's DDE callback function. A transaction is similar to a message—it is a named constant accompanied by other parameters that contain additional information about the transaction.

The DDEML passes a transaction to an application-defined DDE callback function, which carries out the appropriate action depending on the type of the transaction. For example, when a client application attempts to establish a conversation with a server application, the client calls the **DdeConnect** function. This causes the DDEML to send an XTYP_CONNECT transaction to the server's DDE callback function. The callback function can allow the conversation by returning TRUE to the DDEML, or it can deny the conversation by returning FALSE.

For a detailed discussion of transactions, see Section 5.8, "Transaction Management."

5.1.3 Service Names, Topic Names, and Item Names

A DDE server uses a three-level hierarchy—service name (called "application name" in previous DDE documentation), topic name, and item name—to uniquely identify a unit of data that the server can exchange during a conversation. A service name is a string that a server application responds to when a client attempts to establish a conversation with the server. A client must specify this service name to be able to establish a conversation with the server. Although a server can respond to many service names, most servers respond to only one name.

A topic name is a string that identifies a logical data context. For servers that operate on file-based documents, topic names are typically filenames; for other servers, they are other application-specific strings. A client must specify a topic name along with a server's service name when it attempts to establish a conversation with a server.

An item name is a string that identifies a unit of data that a server can pass to a client during a transaction. For example, an item name might identify an integer, a string, several paragraphs of text, or a bitmap.

To a client, these names are the keys that make it possible for the client to establish a conversation with a server and to receive data from the server.

5.1.4 System Topic

The System topic provides a context for information that may be of general interest to any DDE client. Server applications are encouraged to support the System topic at all times. (The System topic is defined in the DDEML header file as SZDDESYS_TOPIC.)

To find out which servers are present and the kinds of information they can provide, a client can request a conversation on the System topic with the service name set to NULL when the client application starts. Such wildcard conversations should be kept to a minimum, because they are costly in terms of system performance.

For more information about initiating DDE conversations, see Section 5.6, "Conversation Management."

A server should support the following item names within the System topic and any other item names that may be useful to a client:

Item	Description
SZDDE_ITEM_ITEMLIST	A list of the items that are supported under a non-System topic. (This list may vary from moment to moment and from topic to topic.)
SZDDESYS_ITEM_FORMATS	A list of clipboard format numbers that the server can render. This list should be ordered with the most descriptive formats first. A server may not be able to render all items in all formats within this list. At a minimum, a server should support the CF_TEXT clipboard format for item names associated with the System topic. For more information about clipboard formats and rendering data, see the *Microsoft Windows Guide to Programming*.
SZDDESYS_ITEM_HELP	General help information.
SZDDESYS_ITEM_RTNMSG	Supporting detail for the most recently used WM_DDE_ACK message. This is useful when more than 8 bits of application-specific return data are required.
SZDDESYS_ITEM_STATUS	An indication of the current status of the server. Typically, this item supports only the CF_TEXT format and contains the Ready or Busy string.
SZDDESYS_ITEM_SYSITEMS	A list of the items supported under the System topic by this server.
SZDDESYS_ITEM_TOPICS	A list of the topics supported by the server at the current time. (This list may vary from moment to moment.)

These item names are string constants defined in the DDEML header files. To obtain string handles for these strings, an application must use the DDEML string-management functions, just as it would for any other string in a DDEML application. For more information about managing strings, see Section 5.4, "String Management."

5.2 Initialization

The DDEML requires that Windows be running; otherwise, the system cannot load the DDEML dynamic-link library. Before calling any DDEML function, an application should call the **GetWinFlags** function, checking the return value for the WF_PMODE flag. If this flag is returned, the application can call DDEML functions.

Before calling any other DDEML function, an application must call the **Dde-Initialize** function. The **DdeInitialize** function obtains an instance identifier for the application, registers the application's DDE callback function with the DDEML, and specifies the transaction filter flags for the callback function.

The DDEML uses instance identifiers so that it can support applications that allow multiple DDEML instances. Each instance of an application must pass its instance identifier as the *idInst* parameter to any other DDEML function that requires it. An application that uses multiple DDEML instances should assign a different DDE callback function to each instance. This makes it possible for the application to identify each instance within its callback function.

The purpose for multiple DDEML instances is to support DLLs using the DDEML. It is not recommended that an application have multiple DDE instances.

Transaction filters optimize system performance by preventing the DDEML from passing unwanted transactions to the application's DDE callback function. An application sets the transaction filters when it calls the **DdeInitialize** function. An application should specify a transaction filter flag for each type of transaction that it does not process in its callback function. An application can change its transaction filters with a subsequent call to the **DdeInitialize** function. For a complete list of transaction filter flags, see the description of the **DdeInitialize** function in the *Microsoft Windows Programmer's Reference, Volume 2.*

For more information about transactions, see Section 5.8, "Transaction Management."

The following example shows how to initialize an application to use the DDEML:

```
DWORD idInst = 0L;  /* instance identifier       */
HANDLE hInst;       /* instance handle           */
FARPROC lpDdeProc;  /* procedure instance address */

lpDdeProc = MakeProcInstance((FARPROC) DdeCallback, hInst);
if (DdeInitialize(&idInst,       /* receives instance identifier    */
        (PFNCALLBACK) lpDdeProc, /* address of callback function    */
        CBF_FAIL_EXECUTES |      /* filter XTYP_EXECUTE transactions */
        CBF_FAIL_POKES, 0L);     /* filter XTYP_POKE transactions   */
    return FALSE;
```

This example obtains a procedure-instance address for the callback function named **DdeCallback** and then passes the address to the DDEML. The CBF_FAIL_EXECUTES and CBF_FAIL_POKES filters prevent the DDEML from passing XTYP_EXECUTE or XTYP_POKE transactions to the callback function.

An application should call the **DdeUninitialize** function when it no longer needs to use the DDEML. This function terminates any conversations currently open for the application and frees the DDEML resources that the system allocated for the application.

The DDEML may have difficulty terminating a conversation. This occurs when the other partner in a conversation fails to terminate its end of the conversation. In this case, the system enters a modal loop while it waits for any conversations to be terminated. A system-defined timeout period is associated with this loop. If the timeout period expires before the conversations have been terminated, a message box appears that gives the user the choice of waiting for another timeout period (Retry), waiting indefinitely (Ignore), or exiting the modal loop (Abort). An application should call **DdeUninitialize** after it has become invisible to the user and after its message loop has terminated.

5.3 Callback Function

An application that uses the DDEML must provide a callback function that processes the DDE events that affect the application. The DDEML notifies an application of such events by sending transactions to the application's DDE callback function. The transactions that a callback function receives depend on the callback-filter flags that the application specified in the **DdeInitialize** function and on whether the application is a client, a server, or both. The following example shows the general structure of a callback function for a typical client application:

```
HDDEDATA EXPENTRY DdeCallback(wType, wFmt, hConv, hsz1,
    hsz2, hData, dwData1, dwData2)
WORD wType;         /* transaction type              */
WORD wFmt;          /* clipboard format              */
HCONV hConv;        /* handle of the conversation    */
HSZ hsz1;           /* handle of a string            */
HSZ hsz2;           /* handle of a string            */
HDDEDATA hData;     /* handle of a global memory object */
DWORD dwData1;      /* transaction-specific data     */
DWORD dwData2;      /* transaction-specific data     */
{
    switch (wType) {
        case XTYP_REGISTER:
        case XTYP_UNREGISTER:
            .
            .
            .
            return (HDDEDATA) NULL;
        case XTYP_ADVDATA:
            .
            .
            .
            return (HDDEDATA) DDE_FACK;

        case XTYP_XACT_COMPLETE:
            .
            .
            .
            return (HDDEDATA) NULL;

        case XTYP_DISCONNECT:
            .
            .
            .
            return (HDDEDATA) NULL;

        default:
            return (HDDEDATA) NULL;
    }
}
```

The *wType* parameter specifies the transaction type sent to the callback function by the DDEML. The values of the remaining parameters depend on the transaction type. The transaction types and the events that generate them are described in the following sections of this chapter. For detailed information about each transaction type, see Section 5.8, "Transaction Management."

5.4 String Management

Many DDEML functions require access to strings in order to carry out a DDE task. For example, a client must specify a service name and a topic name when it calls the **DdeConnect** function to request a conversation with a server. An application specifies a string by passing a string handle rather than a pointer in a DDEML function. A string handle is a doubleword value, assigned by the system, that identifies a string.

An application can obtain a string handle for a particular string by calling the **DdeCreateStringHandle** function. This function registers the string with the system and returns a string handle to the application. The application can pass the handle to DDEML functions that need to access the string. The following example obtains string handles for the System topic string and the service-name string:

```
HSZ hszServName;
HSZ hszSysTopic;

hszServName = DdeCreateStringHandle(
    idInst,         /* instance identifier */
    "MyServer",     /* string to register  */
    CP_WINANSI);    /* code page           */

hszSysTopic = DdeCreateStringHandle(
    idInst,         /* instance identifier */
    SZDDESYS_TOPIC, /* System topic        */
    CP_WINANSI);    /* code page           */
```

The *idInst* parameter in the preceding example specifies the instance identifier obtained by the **DdeInitialize** function.

An application's DDE callback function receives one or more string handles during most DDE transactions. For example, a server receives two string handles during the XTYP_REQUEST transaction: One identifies a string specifying a topic name; the other identifies a string specifying an item name. An application can obtain the length of the string that corresponds to a string handle and copy the string to an application-defined buffer by calling the **DdeQueryString** function, as the following example demonstrates:

```
DWORD idInst;
DWORD cb;
HSZ hszServ;
PSTR pszServName;

cb = DdeQueryString(idInst, hszServ, (LPSTR) NULL, 0L, CP_WINANSI) + 1;
pszServName = (PSTR) LocalAlloc(LPTR, (WORD) cb);
DdeQueryString(idInst, hszServ, pszServName, cb, CP_WINANSI);
```

An instance-specific string handle is not mappable from string handle to string to string handle again. For instance, in the following example, the **DdeQueryString** function creates a string from a string handle and then **DdeCreateStringHandle** creates a string handle from that string, but the two handles are not the same:

```
DWORD cb;
HSZ hszInst, hszNew;
PSZ pszInst;

DdeQueryString(idInst, hszInst, pszInst, cb, CP_WINANSI);
hszNew = DdeCreateStringHandle(idInst, pszInst, CP_WINANSI);
/* hszNew != hszInst ! */
```

A string handle that is passed to an application's DDE callback function becomes invalid when the callback function returns. An application can save a string handle for use after the callback function returns by using the **DdeKeepStringHandle** function.

When an application calls **DdeCreateStringHandle**, the system enters the specified string into a systemwide string table and generates a handle that it uses to access the string. The system also maintains a usage count for each string in the string table.

When an application calls the **DdeCreateStringHandle** function and specifies a string that already exists in the table, the system increments the usage count rather than adding another occurrence of the string. (An application can also increment the usage count by using the **DdeKeepStringHandle** function.) When an application calls the **DdeFreeStringHandle** function, the system decrements the usage count.

A string is removed from the table when its usage count equals zero. Because more than one application can obtain the handle of a particular string, an application should not free a string handle more times than it has created or kept the handle. Otherwise, the application could cause the string to be removed from the table, denying other applications access to the string.

The DDEML string-management functions are based on the Windows atom manager and are subject to the same size restrictions as atoms.

5.5 Name Service

The DDEML makes it possible for a server application to register the service names that it supports and to prevent the DDEML from sending XTYP_CONNECT transactions for unsupported service names to the server's DDE callback function. The remaining topics in this section describe this service.

5.5.1 Service-Name Registration

By registering its service names with the DDEML, a server informs other DDE applications in the system that a new server is available. A server registers a service name by calling the **DdeNameService** function, specifying a string handle that identifies the name. As a result, the DDEML sends an XTYP_REGISTER transaction to the callback function of each DDEML application in the system (except those that specified the CBF_SKIP_REGISTRATIONS filter flag in the **DdeInitialize** function). The XTYP_REGISTER transaction passes two string handles to a callback function: The first identifies the string specifying the base service name; the second identifies the string specifying the instance-specific service. A client typically uses the base service name in a list of available servers, so that the user can select a server from the list. The client uses the instance-specific service name to establish a conversation with a specific instance of a server application if more than one instance is running.

A server can use the **DdeNameService** function to unregister a service name. This causes the DDEML to send XTYP_UNREGISTER transactions to the other DDE applications in the system, informing them that they can no longer use the name to establish conversations.

A server should call the **DdeNameService** function to register its service names soon after calling the **DdeInitialize** function. A server should unregister its service names just before calling the **DdeUninitialize** function.

5.5.2 Service-Name Filter

Besides registering service names, the **DdeNameService** function makes it possible for a server to turn its service-name filter on or off. When a server turns off its service-name filter, the DDEML sends the XTYP_CONNECT transaction to the server's DDE callback function whenever any client calls the **DdeConnect** function, regardless of the service name specified in the function. When a server turns on its service-name filter, the DDEML sends the XTYP_CONNECT transaction to the server only when the **DdeConnect** function specifies a service name that the server has specified in a call to the **DdeNameService** function.

By default, the service-name filter is on when an application calls **DdeInitialize**. This prevents the DDEML from sending the XTYP_CONNECT transaction to a server before the server has created the string handles that it needs. A server can turn off its service-name filter by specifying the DNS_FILTEROFF flag in a call to the **DdeNameService** function. The DNS_FILTERON flag turns on the filter.

5.6 Conversation Management

A conversation between a client and a server is always established at the request of the client. When a conversation is established, each partner receives a handle that identifies the conversation. The partners use this handle in other DDEML functions to send transactions and manage the conversation.

A client can request a conversation with a single server, or it can request multiple conversations with one or more servers. The remaining topics in this section describe how an application establishes conversations and explain how an application can obtain information about conversations that are already established.

5.6.1 Single Conversations

A client application requests a single conversation with a server by calling the **DdeConnect** function, specifying string handles that identify the strings specifying the service name of the server and the topic name of interest. The DDEML responds by sending the XTYP_CONNECT transaction to the DDE callback function of each server application that either has registered a service name that matches the one specified in the **DdeConnect** function or has turned service-name filtering off by calling the **DdeNameService** function. A server can also filter the XTYP_CONNECT transactions by specifying the CBF_FAIL_CONNECTIONS filter flag in the **DdeInitialize** function. During the XTYP_CONNECT transaction, the DDEML passes the service name and the topic name to the server. The server should examine the names and return TRUE if it supports the service/topic name pair or FALSE if it does not.

If no server returns TRUE from the XTYP_CONNECT transaction, the client receives NULL from the **DdeConnect** function and no conversation is established. If a server does return TRUE, a conversation is established and the client receives a conversation handle—a doubleword value that identifies the conversation. The client uses the handle in subsequent DDEML calls to obtain data from the server. The server receives the XTYP_CONNECT_CONFIRM transaction (unless the server specified the CBF_FAIL_CONFIRMS filter flag). This transaction passes the conversation handle to the server.

The following example requests a conversation on the System topic with a server that recognizes the service name MyServer. The *hszServName* and *hszSysTopic* parameters are previously created string handles.

```
HCONV hConv;
HWND hwndParent;
HSZ hszServName;
HSZ hszSysTopic;

hConv = DdeConnect(
    idInst,         /* instance identifier              */
    hszServName,    /* service-name string handle       */
    hszSysTopic,    /* System-topic string handle       */
    (PCONVCONTEXT) NULL); /* reserved--must be NULL     */

if (hConv == NULL) {
    MessageBox(hwndParent, "MyServer is unavailable.",
        (LPSTR) NULL, MB_OK);
    return FALSE;
}
```

The **DdeConnect** function in the preceding example causes the DDE callback function of the MyServer application to receive an XTYP_CONNECT transaction.

In the following example, the server responds to the XTYP_CONNECT transaction by comparing the topic-name string handle that the DDEML passed to the server with each element in the array of topic-name string handles that the server supports. If the server finds a match, it establishes the conversation.

```
#define CTOPICS 5

HSZ hsz1;                  /* string handle passed by DDEML  */
HSZ ahszTopics[CTOPICS];   /* array of supported topics      */
int i;                     /* loop counter                   */

.
. /* Use switch statement to examine transaction types. */
.

case XTYP_CONNECT:
    for (i = 0; i < CTOPICS; i++) {
        if (hsz1 == ahszTopics[i])
            return TRUE;   /* establish a conversation       */
    }

    return FALSE; /* topic not supported; deny conversation  */

.
. /* Process other transaction types. */
.
```

If the server returns TRUE in response to the XTYP_CONNECT transaction, the DDEML sends an XTYP_CONNECT_CONFIRM transaction to the server's DDE callback function. The server can obtain the handle for the conversation by processing this transaction.

A client can establish a wildcard conversation by specifying NULL for the service-name string handle, the topic-name string handle, or both in a call to the **DdeConnect** function. When at least one of the string handles is NULL, the DDEML sends the XTYP_WILDCONNECT transaction to the callback functions of all DDE applications (except those that filter the XTYP_WILDCONNECT transaction). Each server application should respond by returning a data handle that identifies a null-terminated array of **HSZPAIR** structures. If the server application has not called the **DdeNameService** function to register its service names and filtering is on, the server does not receive XTYP_WILDCONNECT transactions. For more information about data handles, see Section 5.7, "Data Management."

The array should contain one structure for each service/topic name pair that matches the pair specified by the client. The DDEML selects one of the pairs to establish a conversation and returns to the client a handle that identifies the conversation. The DDEML sends the XTYP_CONNECT_CONFIRM transaction to the server (unless the server filters this transaction). The following example shows a typical server response to the XTYP_WILDCONNECT transaction:

```
#define CTOPICS 2

UINT type;
UINT fmt;
HSZPAIR ahp[(CTOPICS + 1)];
HSZ ahszTopicList[CTOPICS];
HSZ hszServ, hszTopic;
WORD i, j;

if (type == XTYP_WILDCONNECT) {

     /*
      * Scan the topic list, and create array of HSZPAIR
      * structures.
      */

    j = 0;
    for (i = 0; i < CTOPICS; i++) {
        if (hszTopic == (HSZ) NULL ||
                hszTopic == ahszTopicList[i]) {
            ahp[j].hszSvc = hszServ;
            ahp[j++].hszTopic = ahszTopicList[i];
        }
    }

    /*
     * End the list with an HSZPAIR structure that contains NULL
     * string handles as its members.
     */

    ahp[j].hszSvc = NULL;
    ahp[j++].hszTopic = NULL;

    /*
     * Return a handle to a global memory object containing the
     * HSZPAIR structures.
     */

    return DdeCreateDataHandle(
        idInst,          /* instance identifier    */
        &ahp,            /* points to HSZPAIR array */
        sizeof(HSZ) * j, /* length of the array    */
        0,               /* start at the beginning  */
        NULL,            /* no item-name string    */
        fmt,             /* return the same format  */
        0);              /* let the system own it   */
}
```

Either the client or the server can terminate a conversation at any time by calling the **DdeDisconnect** function. This causes the callback function of the partner in the conversation to receive the XTYP_DISCONNECT transaction (unless the partner specified the CBF_SKIP_DISCONNECTS filter flag). Typically, an application responds to the XTYP_DISCONNECT transaction by using the **DdeQueryConvInfo** function to obtain information about the conversation that terminated. After the callback function returns from processing the XTYP_DISCONNECT transaction, the conversation handle is no longer valid.

A client application that receives an XTYP_DISCONNECT transaction in its DDE callback function can attempt to reestablish the conversation by calling the **DdeReconnect** function. The client must call **DdeReconnect** from within its DDE callback function.

5.6.2 Multiple Conversations

A client application can use the **DdeConnectList** function to determine whether any servers of interest are available in the system. A client specifies a service name and topic name when it calls the **DdeConnectList** function, causing the DDEML to broadcast the XTYP_WILDCONNECT transaction to the DDE callback functions of all servers that match the service name (except those that filter the transaction). A server's callback function should return a data handle that identifies a null-terminated array of **HSZPAIR** structures. The array should contain one structure for each service/topic name pair that matches the pair specified by the client. The DDEML establishes a conversation for each **HSZPAIR** structure filled by the server and returns a conversation-list handle to the client. The server receives the conversation handle by way of the XTYP_CONNECT_CONFIRM transaction (unless the server filters this transaction).

A client can specify NULL for the service name, topic name, or both when it calls the **DdeConnectList** function. If the service name is NULL, all servers in the system that support the specified topic name respond. A conversation is established with each responding server, including multiple instances of the same server. If the topic name is NULL, a conversation is established on each topic recognized by each server that matches the service name.

A client can use the **DdeQueryNextServer** and **DdeQueryConvInfo** functions to identify the servers that respond to the **DdeConnectList** function. The **DdeQueryNextServer** function returns the next conversation handle in a conversation list; the **DdeQueryConvInfo** function fills a **CONVINFO** structure with information about the conversation. The client can keep the conversation handles that it needs and discard the rest from the conversation list.

The following example uses the **DdeConnectList** function to establish conversations with all servers that support the System topic and then uses the **DdeQueryNextServer** and **DdeQueryConvInfo** functions to obtain the servers' service-name string handles and store them in a buffer:

```
HCONVLIST hconvList; /* conversation list       */
DWORD idInst;        /* instance identifier     */
HSZ hszSystem;       /* System topic            */
HCONV hconv = NULL;  /* conversation handle     */
CONVINFO ci;         /* holds conversation data */
UINT cConv = 0;      /* count of conv. handles  */
HSZ *pHsz, *aHsz;    /* point to string handles */

/* Connect to all servers that support the System topic. */

hconvList = DdeConnectList(idInst, NULL, hszSystem, NULL, NULL);

/* Count the number of handles in the conversation list. */

while ((hconv = DdeQueryNextServer(hconvList, hconv)) != NULL) cConv++;

/* Allocate a buffer for the string handles. */

hconv = NULL;
aHsz = (HSZ *) LocalAlloc(LMEM_FIXED, cConv * sizeof(HSZ));

/* Copy the string handles to the buffer. */

pHsz = aHsz;
while ((hconv = DdeQueryNextServer(hconvList, hconv)) != NULL) {
    DdeQueryConvInfo(hconv, QID_SYNC, (PCONVINFO) &ci);
    DdeKeepStringHandle(idInst, ci.hszSvcPartner);
    *pHsz++ = ci.hszSvcPartner;
}

.
. /* Use the handles; converse with servers. */
.

/* Free the memory, and terminate conversations. */

LocalFree((HANDLE) aHsz);
DdeDisconnectList(hconvList);
```

An application can terminate an individual conversation in a conversation list by calling the **DdeDisconnect** function. An application can terminate all conversations in a conversation list by calling the **DdeDisconnectList** function. Both functions cause the DDEML to send XTYP_DISCONNECT transactions to each partner's DDE callback function. The **DdeDisconnectList** function sends a XTYP_DISCONNECT transaction for each conversation handle in the list.

A client can use the **DdeConnectList** function to enumerate the conversation handles in a conversation list by passing an existing conversation-list handle to the **DdeConnectList** function. The enumeration process removes the handles of terminated conversations from the list.

If the **DdeConnectList** function specifies an existing conversation-list handle and a service name or topic name that is different from those used to create the existing conversation list, the function creates a new conversation list that contains the handles of any new conversations and the handles from the existing list.

The **DdeConnectList** function attempts to prevent duplicate conversations in a conversation list. A duplicate conversation is a second conversation with the same server on the same service name and topic name. Two such conversations would have different handles, yet they would be duplicate conversations.

5.7 Data Management

Because DDE uses global memory to pass data from one application to another, the DDEML provides a set of functions that DDE applications can use to create and manage global memory objects.

All transactions that involve the exchange of data require the application supplying the data to create a local buffer containing the data and then to call the **DdeCreateDataHandle** function. This function allocates a global memory object, copies the data from the buffer to the memory object, and returns a data handle of the application. A data handle is a doubleword value that the DDEML uses to provide access to data in the global memory object. To share the data in a global memory object, an application passes the data handle to the DDEML, and the DDEML passes the handle to the DDE callback function of the application that is receiving the data transaction.

The following example shows how to create a global memory object and obtain a handle of the object. During the XTYP_ADVREQ transaction, the callback function converts the current time to an ASCII string, copies the string to a local buffer, then creates a global memory object that contains the string. The callback function returns the handle of the global memory object to the DDEML, which passes the handle to the client application.

```
typedef struct { /* tm */
    int hour;
    int minute;
    int second;
} TIME;

TIME tmTime;
HSZ hszTime;
HSZ hszNow;
```

```
HDDEDATA EXPENTRY DdeProc(wType, wFmt, hConv, hsz1, hsz2,
    hData, dwData1, dwData2)
WORD wType;
WORD wFmt;
HCONV hConv;
HSZ hsz1;
HSZ hsz2;
HDDEDATA hData;
DWORD dwData1;
DWORD dwData2;
{
    char szBuf[32];

    switch (wType) {

        case XTYP_ADVREQ:
            if ((hsz1 == hszTime && hsz2 == hszNow)
                    && (wFmt == CF_TEXT)) {

                /* Copy formatted time string to buffer. */

                itoa(tmTime.hour, szBuf, 10);
                strcat(szBuf, ":");
                if (tmTime.minute < 10)
                    strcat(szBuf, "0");
                itoa(tmTime.minute, &szBuf[strlen(szBuf)], 10);
                strcat(szBuf, ":");
                if (tmTime.second < 10)
                    strcat(szBuf, "0");
                itoa(tmTime.second, &szBuf[strlen(szBuf)], 10);
                szBuf[strlen(szBuf)] = '\0';

                /* Create global object, and return data handle. */

                return (DdeCreateDataHandle(
                    idInst,            /* instance identifier   */
                    (LPBYTE) szBuf,    /* source buffer         */
                    strlen(szBuf) + 1, /* size of global object */
                    0L,                /* offset from beginning */
                    hszNow,            /* item-name string      */
                    CF_TEXT,           /* clipboard format      */
                    0));               /* no creation flags     */
            } else
                return (HDDEDATA) NULL;

        .
        . /* Process other transaction types. */
        .
    }
}
```

The receiving application obtains a pointer to the global memory object by passing the data handle to the **DdeAccessData** function. The pointer returned by **DdeAccessData** provides read-only access. The application should use the pointer to review the data and then call the **DdeUnaccessData** function to invalidate the pointer. The application can copy the data to a local buffer by using the **DdeGetData** function.

The following example obtains a pointer to the global memory object identified by the *hData* parameter, copies the contents to a local buffer, and then invalidates the pointer:

```
HDDEDATA hData;
LPBYTE lpszAdviseData;
DWORD cbDataLen;
DWORD i;
char szData[32];

case XTYP_ADVDATA:

    lpszAdviseData = DdeAccessData(hData, &cbDataLen);
    for (i = 0; i < cbDataLen; i++)
        szData[i] = *lpszAdviseData++;
    DdeUnaccessData(hData);
    return (HDDEDATA) TRUE;
```

Usually, when an application that created a data handle passes that handle to the DDEML, the handle becomes invalid in the creating application. This is fine if the application needs to share data with just a single application. If an application needs to share the same data with multiple applications, however, the creating application should specify the HDATA_APPOWNED flag in **DdeCreateDataHandle**. Doing so gives ownership of the memory object to the creating application and prevents the DDEML from invalidating the data handle. When the creating application finishes using a memory object it owns, it should free the object by calling the **DdeFreeDataHandle** function.

If an application has not yet passed the handle of a global memory object to the DDEML, the application can add data to the object or overwrite data in the object by using the **DdeAddData** function. Typically, an application uses **DdeAddData** to fill an uninitialized global memory object. After an application passes a data handle to the DDEML, the global memory object identified by the handle cannot be changed; it can only be freed.

The DDEML data-management functions can handle huge memory objects. A DDEML application should check the size of a global memory object and allocate a huge buffer of the appropriate size before copying the object.

5.8 Transaction Management

After a client has established a conversation with a server, the client can send transactions to obtain data and services from the server. The remaining topics in this section describe the types of transactions that clients can use to interact with a server.

5.8.1 Request Transaction

A client application can use the XTYP_REQUEST transaction to request a data item from a server application. The client calls the **DdeClientTransaction** function, specifying XTYP_REQUEST as the transaction type and specifying the data item the application needs.

The DDEML passes the XTYP_REQUEST transaction to the server, specifying the topic name, item name, and data format requested by the client. If the server supports the requested topic, item, and data format, the server should return a data handle that identifies the current value of the item. The DDEML passes this handle to the client as the return value from the **DdeClientTransaction** function. The server should return NULL if it does not support the topic, item, or format requested.

The **DdeClientTransaction** function uses the *lpdwResult* parameter to return a transaction status flag to the client. If the server does not process the XTYP_REQUEST transaction, **DdeClientTransaction** returns NULL, and *lpdwResult* points to the DDE_FNOTPROCESSED or DDE_FBUSY flag. If the DDE_FNOTPROCESSED flag is returned, the client has no way to determine why the server did not process the transaction.

If a server does not support the XTYP_REQUEST transaction, it should specify the CBF_FAIL_REQUESTS filter flag in the **DdeInitialize** function. This prevents the DDEML from sending this transaction to the server.

5.8.2 Poke Transaction

A client can send unsolicited data to a server by using the **DdeClientTransaction** function to send an XTYP_POKE transaction to a server's callback function.

The client application first creates a buffer that contains the data to send to the server and then passes a pointer to the buffer as a parameter to the **DdeClient-Transaction** function. Alternatively, the client can use the **DdeCreateData-Handle** function to obtain a data handle that identifies the data and then pass the handle to **DdeClientTransaction**. In either case, the client also specifies the topic name, item name, and data format when it calls **DdeClientTransaction**.

The DDEML passes the XTYP_POKE transaction to the server, specifying the topic name, item name, and data format that the client requested. To accept the data item and format, the server should return DDE_FACK. To reject the data, the server should return DDE_FNOTPROCESSED. If the server is too busy to accept the data, the server should return DDE_FBUSY.

When the **DdeClientTransaction** function returns, the client can use the *lpdw-Result* parameter to access the transaction status flag. If the flag is DDE_FBUSY, the client should send the transaction again later.

If a server does not support the XTYP_POKE transaction, it should specify the CBF_FAIL_POKES filter flag in the **DdeInitialize** function. This prevents the DDEML from sending this transaction to the server.

5.8.3 Advise Transaction

A client application can use the DDEML to establish one or more links to items in a server application. When such a link is established, the server sends periodic up-dates about the linked item to the client (typically, whenever the value of the item associated with the server application changes). This establishes an advise loop between the two applications that remains in place until the client ends it.

There are two kinds of advise loops: "hot" and "warm." In a hot advise loop, the server immediately sends a data handle that identifies the changed value. In a warm advise loop, the server notifies the client that the value of the item has changed but does not send the data handle until the client requests it.

A client can request a hot advise loop with a server by specifying the XTYP_ADVSTART transaction type in a call to the **DdeClientTransaction** function. To request a warm advise loop, the client must combine the XTYPF_NODATA flag with the XTYP_ADVSTART transaction type. In either event, the DDEML passes the XTYP_ADVSTART transaction to the server's DDE callback function. The server's DDE callback function should examine the parameters that accompany the XTYP_ADVSTART transaction (including the requested format, topic name, and item name) and then return TRUE to allow the advise loop or FALSE to deny it.

After an advise loop is established, the server application should call the **DdePost-Advise** function whenever the value of the item associated with the requested item name changes. This results in an XTYP_ADVREQ transaction being sent to the server's own DDE callback function. The server's DDE callback function should return a data handle that identifies the new value of the data item. The DDEML then notifies the client that the specified item has changed by sending the XTYP_ADVDATA transaction to the client's DDE callback function.

If the client requested a hot advise loop, the DDEML passes the data handle for the changed item to the client during the XTYP_ADVDATA transaction. Otherwise, the client can send an XTYP_REQUEST transaction to obtain the data handle.

It is possible for a server to send updates faster than a client can process the new data. This can be a problem for a client that must perform long processing operations on the data. In this case, the client should specify the XTYPF_ACKREQ flag when it requests an advise loop. This causes the server to wait for the client to acknowledge that it has received and processed a data item before the server sends the next data item. Advise loops that are established with the XTYPF_ACKREQ flag are more robust with fast servers but may occasionally miss updates. Advise loops established without the XTYPF_ACKREQ flag are guaranteed not to miss updates as long as the client keeps up with the server.

A client can end an advise loop by specifying the XTYP_ADVSTOP transaction type in a call to the **DdeClientTransaction** function.

If a server does not support advise loops, it should specify the CBF_FAIL_ADVISES filter flag in the **DdeInitialize** function. This prevents the DDEML from sending the XTYP_ADVSTART and XTYP_ADVSTOP transactions to the server.

5.8.4 Execute Transaction

A client can use the XTYP_EXECUTE transaction to cause a server to execute a command or series of commands.

To execute a server command, the client first creates a buffer that contains a command string for the server to execute and then passes either a pointer to the buffer or a data handle identifying the buffer when it calls the **DdeClientTransaction** function. Other required parameters include the conversation handle, the item-name string handle, the format specification, and the XTYP_EXECUTE transaction type. When an application creates a data handle for passing execute data, the application must specify NULL for the *hszItem* parameter of the **DdeCreateDataHandle** function.

The DDEML passes the XTYP_EXECUTE transaction to the server's DDE callback function specifying the format name, conversation handle, topic name, and data handle identifying the command string. If the server supports the command, it should use the **DdeAccessData** function to obtain a pointer to the command string, execute the command, and then return DDE_FACK. If the server does not support the command or cannot complete the transaction, it should return DDE_FNOTPROCESSED. The server should return DDE_FBUSY if it is too busy to complete the transaction.

When the **DdeClientTransaction** function returns, the client can use the *lpdw-Result* parameter to access the transaction status flag. If the flag is DDE_FBUSY, the client should send the transaction again later.

If a server does not support the XTYP_EXECUTE transaction, it should specify the CBF_FAIL_EXECUTES filter flag in the **DdeInitialize** function. Doing so prevents the DDEML from sending this transaction to the server.

5.8.5 Synchronous and Asynchronous Transactions

A client can send either synchronous or asynchronous transactions. In a synchronous transaction, the client specifies a timeout value that indicates the maximum amount of time to wait for the server to process the transaction. The **DdeClientTransaction** function does not return until the server processes the transaction, the transaction fails, or the timeout value expires. The client specifies the timeout value when it calls **DdeClientTransaction**.

During a synchronous transaction, the client enters a modal loop while waiting for the transaction to be processed. The client can still process user input but cannot send another synchronous transaction until the **DdeClientTransaction** function returns.

A client sends an asynchronous transaction by specifying the TIMEOUT_ASYNC flag in the **DdeClientTransaction** function. The function returns after the transaction is begun, passing a transaction identifier to the client. When the server finishes processing the asynchronous transaction, the DDEML sends an XTYP_XACT_COMPLETE transaction to the client. One of the parameters that the DDEML passes to the client during the XTYP_XACT_COMPLETE transaction is the transaction identifier. By comparing this transaction identifier with the identifier returned by the **DdeClientTransaction** function, the client identifies which asynchronous transaction the server has finished processing.

A client can use the **DdeSetUserHandle** function as an aid to processing an asynchronous transaction. This function makes it possible for a client to associate an application-defined doubleword value with a conversation handle and transaction identifier. The client can use the **DdeQueryConvInfo** function during the XTYP_XACT_COMPLETE transaction to obtain the application-defined doubleword value. This saves an application from having to maintain a list of active transaction identifiers.

If a server does not process an asynchronous transaction in a timely manner, the client can abandon the transaction by calling the **DdeAbandonTransaction** function. The DDEML releases all resources associated with the transaction and discards the results of the transaction when the server finishes processing it.

The asynchronous transaction method is provided for applications that must send a high volume of DDE transactions while simultaneously performing a substantial amount of processing, such as calculations. The asynchronous method is also useful in applications that need to stop processing DDE transactions temporarily so they can complete other tasks without interruption. In most other situations, an application should use the synchronous method.

Synchronous transactions are simpler to maintain and faster than asynchronous transactions. However, only one synchronous transaction can be performed at a time, whereas many asynchronous transactions can be performed simultaneously. With synchronous transactions, a slow server can cause a client to remain idle while waiting for a response. Also, synchronous transactions cause the client to enter a modal loop that could bypass message filtering in the application's own message loop.

5.8.6 Transaction Control

An application can suspend transactions to its DDE callback function—either those transactions associated with a specific conversation handle or all transactions regardless of the conversation handle. This is useful when an application receives a transaction that requires lengthy processing. In this case, an application can return CBR_BLOCK to suspend future transactions associated with that transaction's conversation handle, leaving the application free to process other conversations.

When processing is complete, the application calls the **DdeEnableCallback** function to resume transactions associated with the suspended conversation. Calling **DdeEnableCallback** causes the DDEML to resend the transaction that resulted in the application suspending the conversation. Therefore, the application should store the result of the transaction in such a way that it can obtain and return the result without reprocessing the transaction.

An application can suspend all transactions associated with a specific conversation handle by specifying the handle and the EC_DISABLE flag in a call to the **DdeEnableCallback** function. By specifying a NULL handle, an application can suspend all transactions for all conversations.

When a conversation is suspended, the DDEML saves transactions for the conversation in a transaction queue. When the application reenables the conversation, the DDEML removes the saved transactions from the queue, passing each transaction to the appropriate callback function. Even though the capacity of the transaction queue is large, an application should reenable a suspended conversation as soon as possible to avoid losing transactions.

An application can resume usual transaction processing by specifying the EC_ENABLEALL flag in the **DdeEnableCallback** function. For a more controlled resumption of transaction processing, the application can specify the EC_ENABLEONE flag. This removes one transaction from the transaction queue and passes it to the appropriate callback function; after the single transaction is processed, any conversations are again disabled.

5.8.7 Transaction Classes

The DDEML has four classes of transactions. Each class is identified by a constant that begins with the XCLASS_ prefix. The classes are defined in the DDEML header file. The class constant is combined with the transaction-type constant and is passed to the DDE callback function of the receiving application.

A transaction's class determines the return value that a callback function is expected to return if it processes the transaction. The following table shows the return values and transaction types associated with each of the four transaction classes:

Class	Return value	Transaction
XCLASS_BOOL	TRUE or FALSE	XTYP_ADVSTART XTYP_CONNECT
XCLASS_DATA	A data handle, CBR_BLOCK, or NULL	XTYP_ADVREQ XTYP_REQUEST XTYP_WILDCONNECT
XCLASS_FLAGS	A transaction flag: DDE_FACK, DDE_FBUSY, or DDE_FNOTPROCESSED	XTYP_ADVDATA XTYP_EXECUTE XTYP_POKE
XCLASS_NOTIFICATION	None	XTYP_ADVSTOP XTYP_CONNECT_CONFIRM XTYP_DISCONNECT XTYP_ERROR XTYP_REGISTER XTYP_UNREGISTER XTYP_XACT_COMPLETE

5.8.8 Transaction Summary

The following list shows each DDE transaction type, the receiver of each type, and a description of the activity that causes the DDEML to generate each type:

Transaction type	Receiver	Cause
XTYP_ADVDATA	Client	A server responded to an XTYP_ADVREQ transaction by returning a data handle.
XTYP_ADVREQ	Server	A server called the **DdePost-Advise** function, indicating that the value of a data item in an advise loop had changed.
XTYP_ADVSTART	Server	A client specified the XTYP_ADVSTART transaction type in a call to the **DdeClient-Transaction** function.
XTYP_ADVSTOP	Server	A client specified the XTYP_ADVSTOP transaction type in a call to the **DdeClient-Transaction** function.
XTYP_CONNECT	Server	A client called the **DdeConnect** function, specifying a service name and topic name supported by the server.
XTYP_CONNECT_CONFIRM	Server	The server returned TRUE in response to an XTYP_CONNECT or XTYP_WILDCONNECT transaction.
XTYP_DISCONNECT	Client/Server	A partner in a conversation called the **DdeDisconnect** function, causing both partners to receive this transaction.
XTYP_ERROR	Client/Server	A critical error has occurred. The DDEML may not have sufficient resources to continue.
XTYP_EXECUTE	Server	A client specified the XTYP_EXECUTE transaction type in a call to the **DdeClient-Transaction** function.
XTYP_MONITOR	DDE monitoring application	A DDE event occurred in the system. For more information about DDE monitoring applications, see Section 5.10, "Monitoring Applications."

Transaction type	Receiver	Cause
XTYP_POKE	Server	A client specified the XTYP_POKE transaction type in a call to the **DdeClient-Transaction** function.
XTYP_REGISTER	Client/Server	A server application used the **DdeNameService** function to register a service name.
XTYP_REQUEST	Server	A client specified the XTYP_REQUEST transaction type in a call to the **DdeClient-Transaction** function.
XTYP_UNREGISTER	Client/Server	A server application used the **DdeNameService** function to unregister a service name.
XTYP_WILDCONNECT	Server	A client called the **DdeConnect** or **DdeConnectList** function, specifying NULL for the service name, the topic name, or both.
XTYP_XACT_COMPLETE	Client	An asynchronous transaction, sent when the client specified the TIMEOUT_ASYNC flag in a call to the **DdeClient-Transaction** function, has concluded.

5.9 Error Detection

Whenever a DDEML function fails, an application can call the **DdeGetLastError** function to determine the cause of the failure. The **DdeGetLastError** function returns an error value that specifies the cause of the most recent error.

For a list of possible error values for each DDEML function, see the individual function descriptions in the *Microsoft Windows Programmer's Reference, Volume 2.*

5.10 Monitoring Applications

Microsoft Windows DDESpy (DDESPY.EXE) monitors DDE activity in the system. You can use DDESpy as a tool for debugging your DDE applications. For more information about DDESpy, see *Microsoft Windows Programming Tools.*

You can use the API elements of the DDEML to create your own DDE monitoring applications. Like any DDEML application, a DDE monitoring application contains a DDE callback function. The DDEML notifies a monitoring application's DDE callback function whenever a DDE event occurs, passing information about the event to the callback function. The application typically displays the information in a window or writes it to a file.

To receive notifications from the DDEML, an application must have registered itself as a DDE monitor by specifying the APPCLASS_MONITOR flag in a call to the **DdeInitialize** function. In this same call, the application can specify one or more monitor flags to indicate the types of events of which the DDEML is to notify the application's callback function. The following table describes each of the monitor flags an application can specify:

Flag	Meaning
MF_CALLBACKS	Notifies the callback function whenever a transaction is sent to any DDE callback function in the system.
MF_CONV	Notifies the callback function whenever a conversation is established or terminated.
MF_ERRORS	Notifies the callback function whenever a DDEML error occurs.
MF_HSZ_INFO	Notifies the callback function whenever a DDEML application creates, frees, or increments the use count of a string handle or whenever a string handle is freed as a result of a call to the **DdeUninitialize** function.
MF_LINKS	Notifies the callback function whenever an advise loop is started or ended.
MF_POSTMSGS	Notifies the callback function whenever the system or an application posts a DDE message.
MF_SENDMSGS	Notifies the callback function whenever the system or an application sends a DDE message.

The following example shows how to register a DDE monitoring application so that its DDE callback function receives notifications of all DDE events:

```
DWORD idInst;
PFNCALLBACK lpDdeProc;
hInst = hInstance;

lpDdeProc = (PFNCALLBACK) MakeProcInstance(
    (FARPROC) DDECallback,  /* points to callback function      */
    hInstance);             /* instance handle                  */
```

```
if (DdeInitialize(
        (LPDWORD) &idInst,    /* instance identifier             */
        lpDdeProc,            /* points to callback function     */
        APPCLASS_MONITOR |    /* this is a monitoring application */
        MF_CALLBACKS     |    /* monitor callback functions      */
        MF_CONV          |    /* monitor conversation data       */
        MF_ERRORS        |    /* monitor DDEML errors            */
        MF_HSZ_INFO      |    /* monitor data-handle activity    */
        MF_LINKS         |    /* monitor advise loops            */
        MF_POSTMSGS      |    /* monitor posted DDE messages     */
        MF_SENDMSGS,          /* monitor sent DDE messages       */
        0L))                  /* reserved                        */
    return FALSE;
```

The DDEML informs a monitoring application of a DDE event by sending an XTYP_MONITOR transaction to the application's DDE callback function. During this transaction, the DDEML passes a monitor flag that specifies the type of DDE event that has occurred and a handle of a global memory object that contains detailed information about the event. The DDEML provides a set of structures that the application can use to extract the information from the memory object. There is a corresponding structure for each type of DDE event. The following table describes each of these structures:

Structure	Description
MONCBSTRUCT	Contains information about a transaction.
MONCONVSTRUCT	Contains information about a conversation.
MONERRSTRUCT	Contains information about the latest DDE error.
MONLINKSTRUCT	Contains information about an advise loop.
MONHSZSTRUCT	Contains information about a string handle.
MONMSGSTRUCT	Contains information about a DDE message that was sent or posted.

The following example shows the DDE callback function of a DDE monitoring application that formats information about each string handle event and then displays the information in a window. The function uses the **MONHSZSTRUCT** structure to extract the information from the global memory object.

```
HDDEDATA CALLBACK DDECallback(wType, wFmt, hConv, hsz1, hsz2,
    hData, dwData1, dwData2)
WORD wType;
WORD wFmt;
HCONV hConv;
HSZ hsz1;
HSZ hsz2;
HDDEDATA hData;
DWORD dwData1;
DWORD dwData2;
```

```
{
    LPVOID lpData;
    char *szAction;
    char buf[256];
    DWORD cb;

    switch (wType) {
        case XTYP_MONITOR:

            /* Obtain a pointer of the global memory object. */

            if (lpData = DdeAccessData(hData, &cb)) {

                /* Examine the monitor flag. */

                switch (dwData2) {
                    case MF_HSZ_INFO:

#define PHSZS ((MONHSZSTRUCT FAR *)lpData)

                        /*
                         * The global memory object contains
                         * string-handle data. Use the MONHSZSTRUCT
                         * structure to access the data.
                         */

                        switch (PHSZS->fsAction) {

                            /*
                             * Examine the action flags to determine
                             * the action performed on the handle.
                             */

                            case MH_CREATE:
                                szAction = "Created";
                                break;

                            case MH_KEEP:
                                szAction = "Incremented";
                                break;

                            case MH_DELETE:
                                szAction = "Deleted";
                                break;

                            case MH_CLEANUP:
                                szAction = "Cleaned up";
                                break;

                            default:
                                DdeUnaccessData(hData);
                                return ((HDDEDATA) 0);
                        }
                        /* Write formatted output to a buffer. */

                        wsprintf(buf,
                            "Handle %s, Task: %x, Hsz: %lx(%s)",
                            (LPSTR) szAction, PHSZS->hTask, PHSZS->hsz,
                            (LPSTR) PHSZS->str);
                        .
                        . /* Display text in window or write to file. */
                        .

                        break;

#undef PHSZS

                    .
                    . /* Process other MF_* flags. */
                    .

                    default:
                        break;
                }
            }

            /* Free the global memory object. */

            DdeUnaccessData(hData);
            break;

        default:
            break;
    }
    return ((HDDEDATA) 0);
}
```

Object Linking and Embedding Libraries

Chapter 6

This chapter describes the implementation of object linking and embedding (OLE) for applications that run with the Microsoft Windows operating system. The chapter also describes how an application can use linked and embedded objects to create compound documents. The following topics are related to the information in this chapter:

- Dynamic data exchange (DDE)
- Clipboard
- Registration database
- Dynamic-link libraries
- Multiple document interface

This chapter does not go into detail about the recommended user interface for applications that use linked and embedded objects. For information about this subject, see *Microsoft Windows User Interface Guidelines*.

6.1 Basics of Object Linking and Embedding

This section explains some basic OLE concepts and compares OLE functionality to that of the Dynamic Data Exchange Management Library (DDEML).

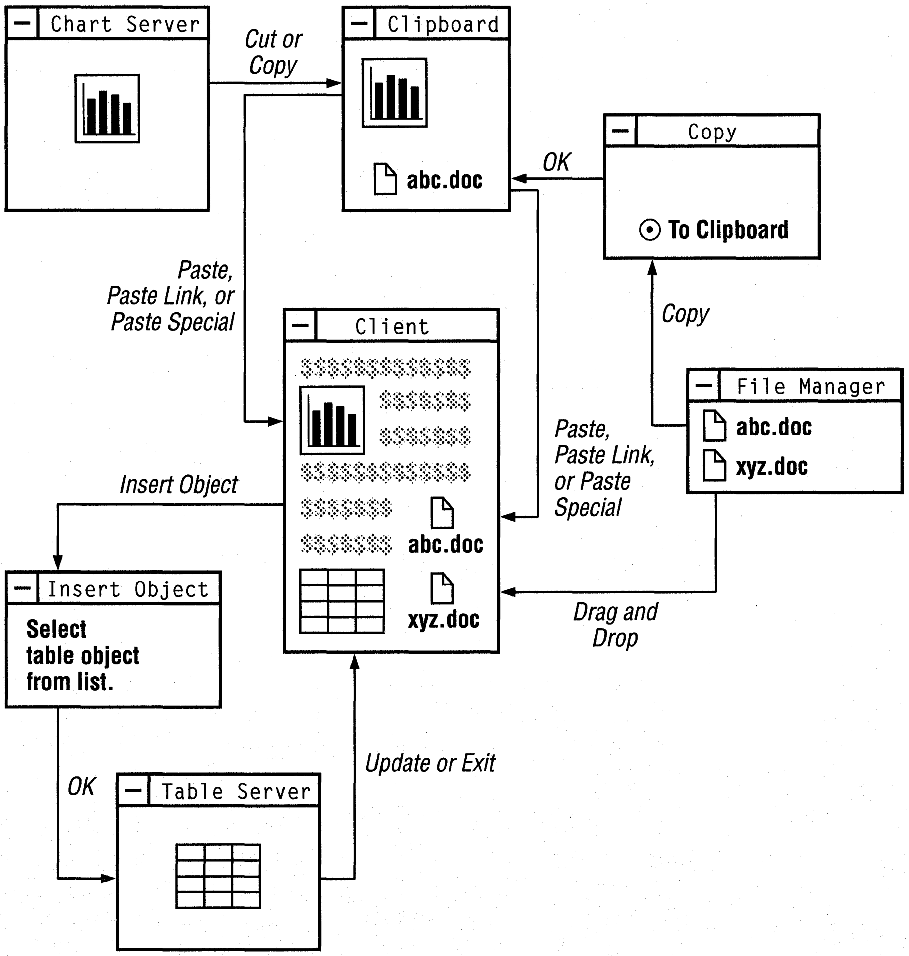
6.1.1 Compound Documents

An application that uses OLE can cooperate with other OLE applications to produce a document containing different kinds of data, all of which can be easily manipulated by the user. The user editing such a document is able to improve the document by using the best features of many different applications. An application that implements OLE gives its users the ability to move away from an application-centered view of computing and toward a document-centered view. In application-centered computing, the tool used to complete a task is often a single application; whereas, in document-centered computing, a user can combine the advantages of many tools to complete a job.

A document that uses linked and embedded objects can contain many kinds of data in many different formats; such a document is called a compound document. A compound document uses the facilities of different OLE applications to manipulate the different kinds of data it displays. Any kind of data format can be incorporated into a compound document; with little or no extra code, OLE applications can even support data formats that have not yet been invented. The user working with a compound document does not need to know which data formats are compatible with one another or how to find and start any applications that created the data. Whenever a user chooses to work with part of a compound document, the application responsible for that part of the document starts automatically.

For example, a compound document could be a brochure that included text, charts, ranges of cells in a spreadsheet, and illustrations. The information could be embedded in the document, or the document could contain links to certain information instead of containing the information itself. The user working with the brochure could automatically switch between the applications that produced its components.

The following illustration shows the relationships between a compound document and its linked and embedded objects.



6.1.2 Linked and Embedded Objects

An object is any data that can be presented in a compound document and manipulated by a user. Anything from a single cell in a spreadsheet to an entire document can be an object. When an object is incorporated into a document, it maintains an association with the application that produced it. That association can be a link, or the object can be embedded in the file.

If the object is linked, the document provides only minimal storage for the data to which the object is linked, and the object can be updated automatically whenever the data in the original application changes. For example, if a range of spreadsheet cells were linked to information in a text file, the data would be stored in some other file and only a link to the data would be saved with the text file.

If an object is embedded, all the data associated with it is saved as part of the file in which it is embedded. If a range of spreadsheet cells were embedded in a text file, the data in the cells would be saved with the text file, including any necessary formulas; the name of the server for the spreadsheet cells would be saved along with this data. The user could select this embedded object while working with the text file, and the spreadsheet application would be started automatically for editing those cells. The presentation and the behavior of the data is the same for a linked and an embedded object.

6.1.2.1 Packages

A package is a type of OLE object that encapsulates another object, a file, or a command line inside a graphic representation (such as an icon or bitmap). When the user double-clicks the graphic object, the OLE libraries activate the object inside the package. The package itself is always an embedded object, not a link. The contents of a package can be an embedded object, a link, or even a file dropped from Windows File Manager.

Packages are useful for presenting compact token views of large files or OLE objects. An application could also use a package as it would use a hyperlink—that is, to connect information in different documents.

Windows version 3.1 includes the application Microsoft Windows Object Packager (PACKAGER.EXE). With Packager, a user can associate a file or data selection with an icon or graphic.

6.1.2.2 Verbs

The types of actions a user can perform on an object are called verbs. Two typical verbs for an object are Play and Edit.

The nature of an object determines its behavior when a user works with it. The most typical use for some objects, such as voice annotations and animated scripts, is to play them. For example, a user could play an animated script by double-clicking it. In this case, Play is the primary verb for the object.

For other objects, the most typical use is to edit them. In the case of text produced by a word processor, for example, the primary verb could be Edit.

The client application typically specifies the primary verb when the user double-clicks an object. However, the server application determines the meaning of that verb. A user can invoke an object's subsidiary verbs by using the *Class Name* Object command or the Links dialog box. For more information about these topics, see Section 6.2.7, "Client User Interface."

The action taken when a user double-clicks a package is that of the primary verb of the object inside the package. The secondary verb for a packaged object is Edit Package; when the user chooses this verb, Packager starts. The user can use Packager to gain access to the secondary verb for the object inside the package.

Many objects support only one verb—for example, an object created by a text editor might support only Edit. If an object supports only one verb, that verb is used no matter what the client application specifies.

For more information about verbs, see Section 6.2.6, "Registration."

6.1.3 Benefits of Object Linking and Embedding

OLE offers the following benefits:

- An application can specialize in performing one job very well. For example, a drawing application that implements OLE does not need any text-editing tools; a user could put text into the drawing and edit that text by using any text editor that supports OLE.
- An application is automatically extensible for future data formats, because the content of an object does not matter to the containing document.
- A user can concentrate on the task instead of on any software required to complete the task.
- A file can be more compact, because linking to objects allows a file to use an object without having to store that object's data.
- A document can be printed or transmitted without using the application that originally produced the document.
- Linked objects in a file can be updated dynamically.

Future implementations of this protocol could take advantage of a wide variety of object types. For example, the user of a voice-recorder application could dictate a comment, package the comment as an object with a visual representation, and embed the graphic as an object in a text file. When a user double-clicked the graphic for this object (a pair of lips, perhaps), the voice-recorder application would play the recorded comment. Linked and embedded objects also lend themselves to implementations such as animated drawings, executable macro scripts, hypertext, and annotations.

6.1.4 Choosing Between OLE and the DDEML

Applications can exchange data by using either OLE or the DDEML. Unless an application has a strong requirement for managing multiple items in a single conversation with another application, the application should use OLE instead of the DDEML.

Both OLE and the DDEML are message-based systems supported by dynamic-link libraries. Developers are encouraged to use these libraries rather than using the underlying message-based protocols. For more information about the message-based OLE protocol, see Section 6.6, "Direct Use of Dynamic Data Exchange."

Unlike OLE, the DDEML supports multiple items per conversation. With OLE, a client needing links to several objects in a document must establish a separate conversation for each object.

OLE offers the following advantages that the DDEML does not:

Advantage	Description
Extensibility to future enhancements	The OLE libraries may be updated in future releases to support new data formats, link tracking, editing without exiting the client application, and other enhancements that will not be immediately available to applications that use the DDEML.
Persistent embedding and linking of objects	The OLE libraries do most of the work of activating objects when an embedded document is reopened, by reestablishing the conversation between a client and server. In contrast, establishing a DDE link (DDE advise loop) is the responsibility of either the user (if the link is not persistent) or of the application (if the link is persistent).
Rendering of common data formats	The OLE libraries assume the burden of rendering common data formats on a display context. DDE applications, however, must do this work themselves.
Server rendering of specialized data formats	The OLE libraries facilitate the rendering of specialized data formats in the client's display context. (The server application or object handler actually performs the rendering.) The client application has to do very little work to render the embedded or linked data in its display context. Such rendering of embedded or linked data is beyond the scope of the DDEML alone.

Advantage	Description
Activating embedded and linked objects	The OLE libraries support activating a server to edit a linked or embedded object or to render data. Activating servers for data rendering and editing is beyond the scope of the DDEML.
Creating objects and links from the clipboard	The OLE libraries do most of the work when an application is using the clipboard to copy and paste links or exchange objects. In contrast, DDE applications must call the Windows clipboard functions directly to perform such operations.
Creating objects and links from files	The OLE libraries provide direct support for using files to exchange data. No DDE protocol is defined for this purpose.

The OLE libraries use DDE messages instead of the DDEML, because the libraries were written before the DDEML was available.

6.1.4.1 Using OLE for Standard DDE Operations

Although most of the OLE application programming interface (API) was designed for linked and embedded objects, it can also be applied to standard DDE items. In particular, an application can use the OLE API to perform the following DDE tasks:

- Initializing conversations based on application and topic names or wildcards.
- Requesting data for named items in negotiated formats from a server.
- Establishing an advise loop—that is, requesting that a DDE server notify the client of changes to the values of specified items and, optionally, that the server send the data when the change occurs.
- Sending data from a server to a client.
- Poking data from a client to a server.
- Sending a DDE command. (This is supported by the **OleExecute** function.)

An OLE client application receives a pointer to an **OLEOBJECT** structure; this structure includes class name, document name, and item name information. These names correspond exactly to DDE counterparts, as follows:

OLE name	DDE name
Class name	Service name (formerly called "application name")
Document name	Topic name
Item name	Item name

The client can use the **OleCreateFromFile** function to make an object and specify all three names. If the client application needs multiple items from the same topic, it must have an **OLEOBJECT** structure for each item, which causes a DDE conversation to be created for each item.

The client library maps OLE functions that work on the **OLEOBJECT** structure to DDE messages as follows:

OLE function	DDE message
OleExecute	WM_DDE_EXECUTE
OleRequestData	WM_DDE_REQUEST
OleSetData	WM_DDE_POKE

Some functions (such as **OleActivate**) are too complicated for this one-to-one mapping of function to DDE message. For these functions, the DDE message depends on the circumstance.

If a client application needs to duplicate the functionality of WM_DDE_ADVISE with OLE, the client must create the link with **olerender_format** for the *renderopt* parameter, specify the required format, and use the **OleGetData** function to retrieve the value when the callback function receives the OLE_CHANGED notification. If more than one item or format is required, the client must create an **OLEOBJECT** structure for each item/format pair. Although this method creates a conversation for each advise transaction, it may be inefficient if the client needs to create many such conversations.

A server application can make itself accessible to DDE by calling the **OleRegisterServer** function to make the System topic available and the **OleRegisterServerDoc** function to make other topics available. When a client connects and asks for an item, the server library calls the **GetObject** function in the server's **OLESERVERDOCVTBL** structure, followed by other server-implemented functions that are appropriate to the client's request. (Usually, the library calls the **GetData** function in the server's **OLEOBJECTVTBL** structure.) As long as the object allocated by the call to **GetObject** has not been released, the server should send a notification when the item has changed, so that the OLE libraries can send data to clients that have sent WM_DDE_ADVISE.

6.1.4.2 Using Both OLE and the DDEML

Some applications may need features supported only by OLE and may also need to use the DDEML to support simultaneous links for many items that are updated frequently. Client applications of this kind can use the OLE libraries to initiate conversations with OLE servers and the DDEML to initiate conversations with DDE servers.

Server applications that need to support both OLE and the DDEML must use different service names (DDE application names) for OLE and DDE conversations; otherwise, the OLE and DDEML libraries cannot determine which library should respond when an initiation request is received. Typically, the application changes the service name for the OLE conversation in this case, because other applications and the user must use the service name for the DDE conversation, but the OLE service name is hidden.

6.2 Data Transfer in Object Linking and Embedding

This section gives a brief overview of how applications share information under OLE. Details of the implementation are given in later sections of this chapter.

Applications use three dynamic-link libraries (DLLs), OLECLI.DLL, OLESVR.DLL, and SHELL.DLL, to implement object linking and embedding. Object linking and embedding is supported by OLECLI.DLL and OLESVR.DLL. The registration database is supported by SHELL.DLL.

6.2.1 Client Applications

An OLE client application can accept, display, and store OLE objects. The objects themselves can contain any kind of data. A client application typically identifies an object by using a distinctive border or other visual cue, as described in *Microsoft Windows User Interface Guidelines*.

6.2.2 Server Applications

An OLE server is any application that can edit an object when the OLE libraries inform it that the user of a client application has selected the object. (Some servers can perform operations on an object other than editing.) When the user double-clicks an object in a client application, the server associated with that object starts and the user works with the object inside the server application. When the server starts, its window is typically sized so that only the object is visible. If the user double-clicks a linked object, the entire linked file is loaded and the linked portion of the file is selected. For embedded objects, the user chooses the Update command from the File menu to save changes to the object and chooses Exit when finished.

Many applications are capable of acting as both clients and servers for linked and embedded objects.

6.2.3 Object Handlers

Some OLE server applications implement an additional kind of OLE library called an object handler. Object handlers are dynamic-link libraries that act as intermediaries between client and server applications. Typically, an object handler is supplied by the developers of a server application as a way of improving performance. For example, an object handler could be used to redraw a changed object if the presentation data for that object could not be rendered by the client library.

6.2.4 Communication Between OLE Libraries

Client applications use functions from the OLE API to inform the client library, OLECLI.DLL, that a user wants to perform an operation on an object. The client library uses DDE messages to communicate with the server library, OLESVR.DLL. The server library is responsible for starting and stopping the server application, directing the interaction with the server's callback functions, and maintaining communication with the client library.

When a server application modifies an embedded object, the server notifies the server library of changes. The server library then notifies the client library, and the client library calls back to the client application, informing it that the changes have occurred. Typically, the client application then forces a repaint of the embedded object in the document file. If the server changes a linked object, the server library notifies the client library that the object has changed and should be redrawn.

6.2.5 Clipboard Conventions

When first embedding or linking an object, OLE client and server applications typically exchange data by using the clipboard. When a server application puts an object on the clipboard, it represents the object with data formats, such as Native data, OwnerLink data, ObjectLink data, and a presentation format. The order in which these formats are put on the clipboard is very important, because the order determines the type of object. For example, if the first format is Native and the second is OwnerLink, client applications can use the data to create an embedded object. If the first format is OwnerLink, however, the data describes a linked object.

Native data completely defines an object for a particular server. The data can be meaningful only to the server application. The client application provides storage for Native data, in the case of embedded objects.

OwnerLink data identifies the owner of a linked or embedded object.

Presentation formats allow the client library to display the object in a document. CF_METAFILEPICT, CF_DIB, and CF_BITMAP are typical presentation formats. Native data can be used as a presentation format, typically when an object handler has been defined for that class of data. Native data cannot be used twice in the definition of an object, however; if the server puts Native and OwnerLink data on the clipboard to describe an embedded object, it cannot use Native data as a presentation format for that object. The ability of object handlers to use Native data as the presentation data accounts for the significance of the order of the formats: the order is the only way to distinguish between an embedded object and a link that uses Native data for its presentation.

ObjectLink data identifies a linked object's class and document and the item that is the source for the linked object. (If the item name specified in the ObjectLink format is NULL, the link refers to the entire server document.)

The following table describes the contents of the ObjectLink, OwnerLink, and Native clipboard formats:

Format name	Contents
ObjectLink	Null-terminated string for class name, null-terminated string for document name, string for item name with two terminating null characters.
OwnerLink	Null-terminated string for class name, null-terminated string for document name, string for item name with two terminating null characters.
Native	Stream of bytes interpreted only by the server application or object-handler library. This format can be unique to the server application and must allow the server to load and work with the object.

Although the ObjectLink and OwnerLink formats contain the same information, the OLE libraries use them differently. The libraries use OwnerLink format to identify the owner of an object (which can be different from the source of the object) and ObjectLink format to identify the source of the data for an object.

The class name in the ObjectLink or OwnerLink format is a unique name for a class of objects that a server supports. Server applications register the class name or names they support in the registration database. (For example, the class name used by Windows Paintbrush™ is PBrush.) An application can use the class name to look up information about a server in the registration database. (For more information about registration, see Section 6.2.6, "Registration.") The document name is typically a fully qualified path that identifies the file containing a document. The item name uniquely identifies the part of a document that is defined as an object. Item names are assigned by server applications; an item name can be any string that the server uses to identify part of a document. Items names cannot contain the forward-slash (/) character.

Data in OwnerLink or ObjectLink format could look like the following example:

```
Microsoft Excel Worksheet\0c:\directry\docname.xls\0R1C1:R5C3\0\0
```

The order in which various data formats are put on the clipboard depends on the type of data being copied to the clipboard and the capabilities of the server application. The following table shows the order of clipboard data formats for four different types of data selections. An object does not necessarily use all of the formats listed for it.

Source selection	Clipboard contents, in order
Embedded object	Native OwnerLink Picture or other presentation format (optional) ObjectLink (included only if the server also supports links)
Linked object	OwnerLink Picture or other presentation format (optional; for linked objects, this can be Native data) ObjectLink
Pictorial data	Application-specific formats Native OwnerLink Picture ObjectLink
Structured data	Structured data formats (if selection is structured data only) Native OwnerLink Picture, text, and so on ObjectLink

Before copying data for an embedded or linked object to the clipboard, a server puts descriptions of the data formats on the clipboard. These data formats are listed in order of their level of description, from most descriptive to least. (For example, Microsoft Word would put rich-text format (RTF) onto the clipboard first, then the CF_TEXT clipboard format.)

When a user chooses the Paste command, the client application queries the formats on the clipboard and uses the first format that is compatible with the destination for the object. Because server applications put data onto the clipboard in order of their fidelity of description, the first acceptable format found by a client application is the best format for it to use. If the client application finds an acceptable format prior to the Native format, it incorporates the data into the target document without making it an embedded object. (For example, a Microsoft Word document would not make an embedded object from clipboard data that was in RTF format. Similarly, structured data or a structured document would be embedded into a drawing application but would be converted into the destination document's native data type if the destination were a worksheet or structured document.) If the client application cannot accept any of the data formats prior to Native and OwnerLink, it uses the Native and OwnerLink formats to make an embedded object and then finds an appropriate presentation format. The destination application may require different formats depending on where the selection is to be placed in the destination document; for example, pasting into a picture frame and pasting into a stream of text could require different formats.

When a user chooses the Paste Link command from the Edit menu, the client application looks for the ObjectLink format on the clipboard and ignores the Native and OwnerLink formats. The ObjectLink format identifies the source class, document, and object. If the application finds the ObjectLink format and a useful presentation format, it uses them to make an OLE link to the source document for the object. If the ObjectLink format is not available, the client application may look for the Link format and create a DDE link. This type of link does not support the OLE protocol.

When an application that does not support OLE copies from an OLE item on the clipboard, it ignores the Native, OwnerLink and ObjectLink formats; the behavior of the copying application does not change.

6.2.6 Registration

The registration database supports linked and embedded objects by providing a systemwide source of information about whether server applications support the OLE protocol, the names of the executable files for these applications, the verbs for classes of objects, and whether an object-handler library exists for a given class of object. For more information about this database, see Chapter 7, "Shell Library."

When a server application is installed, it registers itself as an OLE server with the registration database. (This database is supported by the dynamic-link library SHELL.DLL.) To register itself as an OLE server, a server application records in the database that it supports one or more OLE protocols. The only protocols supported by version 1.*x* of the Microsoft OLE libraries are StdFileEditing and StdExecute. StdFileEditing is the current protocol for linked and embedded objects. StdExecute is used only by applications that support the **OleExecute** function. (A third name, Static, describes a picture than cannot be edited by using standard OLE techniques.)

When a client activates a linked or embedded object, the client library finds the command line for the server in the database, appends the **/Embedding** or **/Embedding***filename* command-line option, and uses the new command line to start the server. Starting the server with either of these options differs from the user starting it directly. Either a slash (/) or a hyphen (-) can precede the word

Embedding. For details about how a server reacts when it is started with these options, see Section 6.3.8, "Opening and Closing Objects."

The entries in the registration database are used whenever an application or library needs information about an OLE server. For example, client applications that support the Insert Object command refer to the database in order to list the OLE server applications that could provide a new object. The client application also uses the registration database to retrieve the name of the server application for the Paste Special dialog box.

6.2.6.1 Registration Database

Applications typically add key and value pairs to the registration database by using Microsoft Windows Registration Editor (REGEDIT.EXE). Applications could also use the registration functions to add this information to the database.

The registration database stores keys and values as null-terminated strings. Keys are hierarchically structured, with the names of the components of the keys separated by backslash characters (\). The class name and server path should be registered for every class the server supports. (This class name must be the same string as the server uses when it calls the **OleRegisterServer** function.) If a class has an object-handler library, it should be registered using the **handler** keyword. An application should also register all the verbs its class or classes support. (An application's verbs must be sequential; for example, if an object supports three verbs, the primary verb is 0 and the other verbs must be 1 and 2.)

To be available for OLE transactions, a server should register the key and value pairs shown in the following example when it is installed. This example shows the form of key and value pairs as they would be added to a database with Registration Editor. Although the text string sometimes wraps to the next line in this example, the lines should not include newline characters when they are added to the database.

HKEY_CLASSES_ROOT\class name = *readable version of class name*
HKEY_CLASSES_ROOT\*.ext* = *class name*
HKEY_CLASSES_ROOT\class name\protocol\StdFileEditing\server =
    executable file name
HKEY_CLASSES_ROOT\class name\protocol\StdFileEditing\handler =
    dll name
HKEY_CLASSES_ROOT\class name\protocol\StdFileEditing\verb\0 =
    primary verb
HKEY_CLASSES_ROOT\class name\protocol\StdFileEditing\verb\1 =
    secondary verb

Servers that support the **OleExecute** function also add the following line to the database:

HKEY_CLASSES_ROOT\class name\\protocol\\StdExecute\\server =
executable file name

An ampersand (&) can be used in the verb specification to indicate that the following character is an accelerator key. For example, if a verb is specified as &Edit, the E key is an accelerator key.

A server can register the entire path for its executable file, rather than registering only the filename and arguments. Registering only the filename fails if the application is installed in a directory that is not mentioned in the PATH environment variable. Usually, registering the path and filename is less ambiguous than registering only the filename.

Servers can register data formats that they accept on calls to the **OleSetData** function or that they can return when a client calls the **OleRequestData** function. Clients can use this information to initialize newly created objects (for example, from data selected in the client) or when using the server as an engine (for example, when sending data to a chart and getting a new picture back). Client applications should not depend on the requested data format, because the calls can be rejected by the server.

In the following example, *format* is the string name of the format as passed to the **RegisterClipboardFormat** function or is one of the system-defined clipboard formats (for example, CF_METAFILEPICT):

HKEY_CLASSES_ROOT\class name\\protocol\\StdFileEditing
\\SetDataFormats = *format*[,*format*]
HKEY_CLASSES_ROOT\class name\\protocol\\StdFileEditing
\\RequestDataFormats = *format*[,*f*ormat]

For compatibility with earlier applications, the system registration service also reads and writes registration information in the [embedding] section of the WIN.INI initialization file.

In the following example, the keyword **picture** indicates that the server can produce metafiles for use when rendering objects:

[embedding]
*classname=comment, textual class name,path/arguments,*picture

6.2.6.2 Version Control for Servers

Server applications should store version numbers in their Native data formats. New versions of servers that are intended to replace old versions should be capable of dealing with data in Native format that was created by older versions. It is sometimes important to give the user the option of saving the data in the old format, to support an environment with a mixture of new and old versions, or to permit data to be read by other applications that can interpret only the old format.

There can be only one application at a time (on one workstation) registered as a server for a given class name. The class name (which is stored with the Native data for objects) and the server application are associated in the registration database when the server application registers during installation.

If a new version of a server application allows the user to keep the old version available, a new class name should be allocated for the new server. A good way to do this is to append a version number to the class name. This allows the user to easily differentiate between the two versions when necessary. (The OLE libraries do not check these numbers.)

When the new version of the server is installed, the user should be given the option of either mapping the old objects to the new server (registering the new server as the server for both class names) or keeping them separate. When the user keeps them separate, the user will be aware of two kinds of object (for example, Graph1 and Graph2).

The user should be able to discard the old server version at a later time by remapping the registration database, typically with the help of the server setup program. To remap the database, the old and new objects are given the same value for *readable version of class name* (although their class names remain distinct). The OLE client library removes duplicate names when it produces the list in the Insert Object dialog box. When a client application produces a list by enumerating the registration database, the application must do this filtering itself.

6.2.7 Client User Interface

When a user opens a document that contains a linked or embedded object, the client application uses the OLE functions to communicate with OLECLI.DLL. This library assists the client application with such tasks as loading and drawing objects, updating objects (when necessary), and interacting with server applications.

6.2.7.1 New and Changed Commands

An OLE client application typically implements the following new or changed commands as part of its Edit menu. (Although this user interface is not mandatory, it is recommended for consistency with existing OLE applications.)

Command	Description
Copy	Copies an object from a document to the clipboard.
Cut	Removes an object from a document and places it on the clipboard.
Paste	Copies an object from the clipboard to a document.
Paste Link	Inserts a link between a document and the file that contains an object.
Class Name Object	Makes it possible for the user to activate the verbs for a linked or embedded object. The actual text used instead of the *Class Name* placeholder depends upon the selected object.
Links	Makes it possible for the user to change link updating options, update linked objects, cancel links, repair broken links, and activate the verbs associated with linked objects.
Insert Object	Starts the server application chosen by the user from a dialog box and embeds in a document the object produced by the server. This command is optional.
Paste Special	Transfers an object from the clipboard to a document or inserts a link to the object, using the data format chosen by the user from a dialog box. This command is optional.

In addition to the listed menu changes, client applications must also implement changes to their Copy and Cut commands. When a linked or embedded object is selected in the client application, the application can use the **OleCopyToClipboard** function to implement the Cut and Copy commands.

When the user chooses the Paste command, a client application should insert the contents of the clipboard at the current position in a document. If the clipboard contains an object, choosing this command typically embeds the object in the document.

When the user chooses the Paste Link command, the client library typically inserts a linked object at the current position in a document. The object is displayed in the document, but the Native data that defines that object is stored elsewhere.

If a user copies a linked object to the clipboard, other documents can use this object to produce a link to the original data.

The *Class Name* Object command allows the user to choose one of an object's verbs. If the selection in the document is an embedded object, the *Class Name* placeholder is typically replaced by the class and name of the object; for example, if a user selects an object that is a range of spreadsheet cells for Microsoft Excel, the text of the command might be "Microsoft Excel Worksheet Object." If an object supports only one verb, the name of the verb should precede the class name in the menu item; for example, if the only verb for a text object is Edit, the text of the command might be "Edit WPDocument Object." When an object supports more than one verb, choosing the *Class Name* Object command brings up a cascading menu listing each of the verbs.

For more information about verbs, see Section 6.1.2.2, "Verbs."

Choosing the Links command brings up a Links dialog box, which lists the selected links and their source documents and gives the user the opportunity to change how the links are updated, cancel the link, change the link, or activate the verbs for the link. A user can use this dialog box to repair links to objects that have been moved or renamed.

When the user chooses the Paste Special command, a client application should bring up a dialog box listing the data formats the client supports that are presently on the clipboard. The Paste Special dialog box makes if possible for the user to override the default behaviors of the Paste and Paste Link commands. For example, if the first format on the clipboard can be edited by the client application, the default behavior is for the client to copy the data into the document without making it into an object. The user could override this default behavior and create an object from such data by using the Paste Special command.

When the user chooses the Insert Object command, a client application should allow the user to insert an object of a specified class at the current position in a document. For example, to insert a range of spreadsheet cells in a text document, a user could choose the Insert Object command and select "Microsoft Excel Worksheet" from the dialog box. Selecting this item would start Microsoft Excel. The user would use Microsoft Excel to create the object to be embedded in the text document. When finished, the user would quit Microsoft Excel; the range of spreadsheet cells would automatically be embedded in the text document.

The Insert Object command is optional because a user could achieve the same results without it, although the procedure is less convenient. To use the same example as that shown in the preceding paragraph, the user could leave the client application, start Microsoft Excel, and use the Microsoft Excel Cut or Copy command to transfer data to the clipboard. After returning to the client application, the user could choose the Paste command to move the data from the clipboard into the text document.

If the user chooses the Undo command after activating an object, all the changes made since the object was last updated (or since the object was activated, if it has not been updated) are discarded and the object returns to its state prior to the update. The Undo command closes the connection to the server.

For more information about these commands, including illustrations of the dialog boxes, see *Microsoft Windows User Interface Guidelines.*

6.2.7.2 Using Packages

A package is an embedded graphical object that contains another object, which can be linked or embedded. For example, a user can package a file in an icon and embed the icon in an OLE document. Most of the packaging capabilities are provided by the dynamic-link library SHELL.DLL.

A user can put a package into an OLE document in a number of different ways:

- Copy a file from File Manager to the clipboard, and then choose the Paste or Paste Link command from the Edit menu in the client application.
- Drag a file from File Manager and drop it in the open window for a document in a client application.
- Select Package from the list of objects in the Insert Object dialog box. This starts Object Packager, with which the user can associate a file or data selection with an icon or graphic. Choosing Update and then Exit from Object Packager's File menu puts the package in the client document.
- Run Packager directly, following the steps outlined in the previous list item.

For information about how a client application should react when a user drops a file from File Manager in the client's window, see the description of the **OleCreateFromFile** function in the *Microsoft Windows Programmer's Reference, Volume 2.*

A user whose system does not include the Windows version 3.1 File Manager can follow these steps to create a package by using Object Packager:

- Copy to the clipboard the data to be packaged.
- Open Object Packager and paste the data into it. (At this point, the user could modify the default icon, the default label identifying the icon, or both.)
- Choose Copy Package from the Object Packager Edit menu to copy the package to the clipboard.
- Choose the Paste command from the Edit menu in the client application to embed the package.

For more information about Object Packager, see Section 6.1.2.1, "Packages," or *Microsoft Windows User Interface Guidelines.*

6.2.8 Server User Interface

A server for linked and embedded objects is any application that can be used to edit an object when the OLE libraries inform it that the user of a client application has activated the object. (Some servers can use verbs other than Edit to work with an object.) Although client applications implement many changes to the user interface to support OLE, the user interface does not change significantly for server applications.

OLE servers typically implement changes to the following commands in the Edit menu. (Although this user interface is not mandatory, it is recommended for consistency with existing OLE applications.)

Command	Description
Cut	Transfers data from the application to the clipboard, deleting the data from the source document. A client application can use this data to create an embedded object.
Copy	Transfers a copy of the data from the application to the clipboard. A client application can use this data to create an embedded object and may be able to establish a link to the source document.

Some menu items change names or behave differently when a server is started as part of activating an object from within a compound document. The exact behavior of the server depends on whether the server supports the multiple document interface (MDI).

6.2.8.1 Updating Objects from Multiple-Instance Servers

When an embedded object is edited or played by a multiple-instance server—that is, a server that does not support the multiple document interface (MDI), the Save command on the File menu should change to Update. (This change does not occur when a server starts for a linked object.) When the user chooses the Update command, the object in the client is updated but the focus remains with the server window. To close the server window, the user chooses the Exit command.

When the user chooses the Save As, New, or Open command, the application should display a warning message asking the user whether to update the object in the compound document before performing the action. The New and Open commands break the link between the client and server applications. The Save As command also breaks the link between the client and server if the server was editing an embedded object.

6.2.8.2 Updating Objects from Single-Instance Servers

The same rules for updating objects from multiple-instance servers apply to single-instance (MDI) servers, with the following differences:

- When the focus in an MDI server changes from a window in which an embedded object was activated to a window in which a document that does not contain an embedded object is being edited, the Update command should change back to Save.
- When the user chooses the New or Open command, the window containing the embedded object remains open. (This eliminates the need to prompt the user to update the object.)

6.2.9 Object Storage Formats

The presentation data in linked or embedded objects can be thought of as a presentation object. A presentation objects can be standard, generic, or NULL. A standard presentation object is used when the format is metafile, bitmap, or device-independent bitmap (DIB). The client library supports the presentation objects, including drawing them. Neither client applications nor object handlers can use the presentation objects; they are solely for the use of the client library.

The following list gives the storage format for strings in OLE. The items appear in the order listed.

Type	Description
LONG	Length of string, including terminating null character.
Variable	Null-terminated stream of bytes.

The following list gives the storage format for the standard presentation object used for linked and embedded objects. The items appear in the order listed.

Type	Description
LONG	OLE version number.
LONG	Format identifier. This value is 5.
Variable	Class string. For standard presentation objects, this string is METAFILEPICT, BITMAP, or DIB.
LONG	Width of object, in MM_HIMETRIC units.
LONG	Height of object, in MM_HIMETRIC units.
LONG	Size of presentation data, in bytes.
Variable	Presentation data.

The following list gives the storage format for the generic presentation object used for linked and embedded objects. Generic objects are used when the clipboard format is other than metafile, bitmap, or DIB. The items appear in the order listed.

Type	Description
LONG	OLE version number.
LONG	Format identifier. This value is 5.
Variable	Class string.
LONG	Clipboard format value. If this value exists, the next item in storage is the size of the presentation data.
LONG	Clipboard format name. This value exists only if the clipboard format value is NULL.
LONG	Size of presentation data, in bytes.
Variable	Presentation data.

The following list gives the storage format for embedded objects. The items appear in the order listed.

Type	Description
LONG	OLE version number.
LONG	Format identifier. This value is 2.
Variable	Class string.
Variable	Topic string.
Variable	Item string.
LONG	Size of Native data, in bytes.
Variable	Native data.
Variable	Presentation object (standard, generic, or NULL).

The following list gives the storage format for linked objects. The items appear in the order listed.

Type	Description
LONG	OLE version number.
LONG	Format identifier. This value is 1.
Variable	Class string.
Variable	Topic string.
Variable	Item string.
Variable	Network name string.
short	Network type.

Type	Description
short	Network driver version number.
LONG	Link update options.
Variable	Presentation object (standard, generic, or NULL).

The following list gives the storage format for static objects. The only difference between the format for static objects and the format for standard presentation objects is the value of the format identifier. The items appear in the order listed.

Type	Description
LONG	OLE version number.
LONG	Format identifier. This value is 3.
Variable	Class string. For static objects, this string is METAFILEPICT, BITMAP, or DIB.
LONG	Width of object, in MM_HIMETRIC units.
LONG	Height of object, in MM_HIMETRIC units.
LONG	Size of presentation data, in bytes.
Variable	Presentation data.

6.3 Client Applications

A client application uses a server application to activate and render an object contained by a compound document. A client application provides storage for embedded objects, such contextual information as the target printer and page position, and a means for the user to activate the object and the server application associated with that object. Client applications also provide ways of putting embedded and linked objects into a document and taking them out again.

Client applications must provide permanent storage for objects in the compound document's file. When an item being saved is an embedded object, the client library stores the object's Native data, the presentation data for the object (for example, a metafile), and the OwnerLink information. When the item being saved is a link to another document, the client library stores the presentation data and the ObjectLink format.

Client applications accommodate asynchronous operations by defining a callback function to which the library sends notifications about current operations. As long as the client continues to dispatch messages, it can react to the notifications being sent to the callback function and to input from the user. For more information about asynchronous operations, see Section 6.3.6, "Asynchronous Operations."

6.3.1 Starting a Client Application

When a client application starts, it should follow these steps:

1. Register the clipboard formats that it requires.
2. Allocate and initialize as many **OLECLIENT** structures as required.
3. Allocate and initialize an **OLESTREAM** structure.

A client application can register the clipboard formats by calling the **RegisterClipboardFormat** function for each format, specifying such formats as Native, OwnerLink, ObjectLink, and any other formats it requires.

A client application uses two structures to receive information from the client library: **OLECLIENT** and **OLESTREAM**.

The **OLECLIENT** structure points to an **OLECLIENTVTBL** structure, which in turn points to a callback function supplied by the client application. The OLE libraries use this callback function to notify the client of any changes to an object. The parameters for the callback function are a pointer to the client structure, a pointer to the relevant object, and a value giving the reason for the notification. Typically, an application creates one **OLECLIENT** structure for each **OLEOBJECT** structure. Having a separate **OLECLIENT** structure for each object allows an application to take object-specific action in response to the OLE_QUERY_PAINT callback notification.

The **OLECLIENT** structure can also point to data that describes the state of an object. This data, when present, is supplied and used only by the client application. The client application allocates a separate **OLECLIENT** structure for each object and stores state information about that object in the structure. Because one argument to the callback function is a pointer to the **OLECLIENT** structure, this is an efficient method of retrieving the object's state information when the callback function is called.

The **OLESTREAM** structure points to an **OLESTREAMVTBL** structure, which is a table of pointers to client-supplied functions for stream input and output. The client libraries use these functions when loading and saving objects. A client can customize functions for particular situations, and a client can make such changes as varying the permanent storage for an object; for example, a client could store an object in a database, instead of in a file with the rest of the document.

The client application should create a pointer to the callback function in the **OLECLIENTVTBL** structure and pointers to the functions in the **OLESTREAMVTBL** structure by using the **MakeProcInstance** function. Callback functions should be exported in the module-definition file.

6.3.2 Opening a Compound Document

To open a compound document, a client application should take the following steps:

1. Register the document with the client library.
2. Load the document data from a file.
3. For each object in the document, call the **OleLoadFromStream** function.
4. List any objects with manual links so that the user can update them. Automatically update any automatic links.

The **OleRegisterClientDoc** function registers a document with the client library and returns a handle that is used in object-creation functions and document-management functions. (This registration does not involve the registration database.)

A client application should call the **OleLoadFromStream** function for each object in the document that will be shown on the screen or otherwise activated. (It is often not necessary to load every object in a document immediately when the document is opened.) Parameters for this function include a pointer to the **OLECLIENT** structure, which is used to locate the client's callback function (and which is sometimes used by the client to store private state information for the object), and a pointer to the **OLESTREAM** structure. The library calls the **Get** function in the **OLESTREAMVTBL** structure to load the object.

6.3.3 Document Management

A client application should notify the library when it opens, closes, saves, or renames a document, or causes a document to revert to a previously saved state. A client application can use the following functions to accomplish these tasks:

Function	Description
OleRegisterClientDoc	Registers an opened document with the library.
OleRenameClientDoc	Informs the library that a document has been renamed.
OleRevertClientDoc	Informs the library that a document has reverted to a previously saved state.
OleRevokeClientDoc	Informs the library that a document should be closed or no longer exists.
OleSavedClientDoc	Informs the library that a document has been saved.

A client application should also maintain a persistent name for each object. This name should be unique within the scope of the client document and should be stored with the document. This name is specified when the object is created and should persist when the document is saved and reopened. When a client uses the

OleRename function to change the name of an object, the new name must also be unique and must be stored with the document.

6.3.4 Saving a Document

A client application should follow these steps to save a document:

1. Save the data for the document in the document's file.
2. For each object in the document, call the **OleSaveToStream** function.
3. When the library confirms that all objects have been saved, call the **OleSaved-ClientDoc** function.

A client application can call the **OleQuerySize** function to determine the size of the buffer required to store an object before calling **OleSaveToStream**.

6.3.5 Closing a Document

A client application should follow these steps to close a document:

1. For each object in the document, call the **OleRelease** function.
2. Use either the **OleRevertClientDoc** or the **OleSavedClientDoc** function to register the current state of the document with the library.
3. When the library confirms that all objects have been closed, call the **OleRevokeClientDoc** function.

6.3.6 Asynchronous Operations

When a client application calls a function that invokes a server application, actions taken by the client and server can be asynchronous. For example, the actions of updating a document and closing a server are asynchronous. Whenever an asynchronous operation begins, the client library returns OLE_WAIT_FOR_RELEASE. When a client application receives this notification, it must wait for the OLE_RELEASE notification before it quits. If the client cannot take further action until the asynchronous operation finishes, it should enter a message-dispatch loop and wait for OLE_RELEASE. Otherwise, it should allow the main message loop to continue dispatching messages so that processing can continue.

An application can run only one asynchronous operation at a time for an object; each asynchronous operation must end with the OLE_RELEASE notification before the next one begins. The client's callback function must receive OLE_RELEASE for all pending asynchronous operations before calling the **OleRevokeClientDoc** function.

Some of the object-creation functions return OLE_WAIT_FOR_RELEASE. The client application can continue to work with the document while waiting for OLE_RELEASE, but some functions (for example, **OleActivate**) cannot be called until the asynchronous operation has been completed.

If an application calls a function for an object before receiving OLE_RELEASE for that object, the function may return OLE_BUSY. The server also returns OLE_BUSY when processing a new request would interfere with the processing of a current request from a client application or user. When a function returns OLE_BUSY, the client application can display a message reporting the busy condition at this point or it can enter a loop to wait for the function to return OLE_OK. (The OLE_QUERY_RETRY notification is also sent to the client's callback function when the server is busy; when the callback function returns FALSE, the transaction with the server is ended.) Note that if the server uses the **OleBlockServer** function to postpone OLE activities, the OLE_QUERY_RETRY notification is not sent to the client.

The following example shows a message-dispatch loop that allows a client application to transact messages while waiting for the OLE_RELEASE notification:

```
while ((olestat = OleQueryReleaseStatus(lpObject)) == OLE_BUSY) {
    if (GetMessage(&msg, NULL, NULL, NULL)) {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
}
if (olestat == OLE_ERROR_OBJECT) {
    .
    .  /* The lpObject parameter is invalid. */
    .
}
else { /* if olestat == OLE_OK */
    .
    . /* The object is released, or the server has terminated. */
    .
}
```

A server application could end unexpectedly while a client is waiting for OLE_RELEASE. In this case, the client library recovers properly only if the client uses the **OleQueryReleaseStatus** function, as shown in the preceding example.

The following table shows which OLE functions can return the OLE_WAIT_FOR_RELEASE or OLE_BUSY value to a client application:

Function	OLE_BUSY	OLE_WAIT_FOR_RELEASE
OleActivate	Yes	Yes
OleClose	Yes	Yes

Function	OLE_BUSY	OLE_WAIT_FOR_RELEASE
OleCopyFromLink	Yes	Yes
OleCreate	No	Yes
OleCreateFromClip	No	Yes
OleCreateFromFile	No	Yes
OleCreateFromTemplate	No	Yes
OleCreateLinkFromClip	No	Yes
OleCreateLinkFromFile	No	Yes
OleDelete	Yes	Yes
OleExecute	Yes	Yes
OleLoadFromStream	No	Yes
OleObjectConvert	Yes	No
OleReconnect	Yes	Yes
OleRelease	Yes	Yes
OleRequestData	Yes	Yes
OleSetBounds	Yes	Yes
OleSetColorScheme	Yes	Yes
OleSetData	Yes	Yes
OleSetHostNames	Yes	Yes
OleSetLinkUpdateOptions	Yes	Yes
OleSetTargetDevice	Yes	Yes
OleUnlockServer	No	Yes
OleUpdate	Yes	Yes

6.3.7 Displaying and Printing Objects

When an object has been loaded and, if necessary, brought up to date, the object can be displayed or printed with the container document. To display an object, the client application should set up the device context and bounding rectangle (ensuring that they use the same mapping mode) and then call the **OleDraw** function. The client application can use the **OleQueryBounds** function to retrieve the size of the bounding rectangle on the target device.

An object handler can be used to draw an object. If an object handler exists for an object, the call to the **OleDraw** function is received and processed by the object handler. If there is no object handler, the client library uses the object's presentation data to display or print the object.

If the presentation data for an object is a metafile, the library periodically sends an OLE_QUERY_PAINT notification to the client's callback function while drawing the object. If the callback function returns FALSE, the **OleDraw** function returns immediately and the drawing is ended. A client could also use the OLE_QUERY_PAINT notification to take some actions within the callback function and then return TRUE to indicate that drawing should continue. Any actions the client takes at this time should not interfere with the drawing operation; for example, the client should not scroll the window.

If the target device for an object changes (for example, when the user changes printers), the client application should call the **OleSetTargetDevice** function. The client should also call **OleSetTargetDevice** whenever an object is created or loaded.

If the size of the presentation rectangle for the object changes (for example, through action by the user) the client application should call the **OleSetBounds** function. After calling **OleSetBounds**, the client should call the **OleUpdate** function to update the object and then **OleDraw** to redisplay it.

6.3.8 Opening and Closing Objects

When the user requests the client application to activate an object, the client should check whether the object is busy by calling the **OleQueryReleaseStatus** function. If the object is busy, the client should either refuse the request to open the object or enter a message-dispatch loop, waiting for the OLE_RELEASE notification.

If the object to be activated is not busy, the client should call the **OleActivate** function. The library notifies the client when the server is open or when an error occurs.

The **OleActivate** function allows the client application to specify whether to display the activated object in a window of the server application. A client might hide the server window if an object is updated automatically.

A client application can use the **OleQueryOpen** function to determine whether a specified object is open. The **OleClose** function allows the client to close an open object. Closing an object terminates the connection with the server. To reestablish a terminated connection between a linked object and an open server, the client can use the **OleReconnect** function. To close an open object and release it from memory, a client application can call the **OleRelease** function.

The first time a client application activates a particular embedded object, the client should call the **OleSetHostNames** function, specifying the string the server window should display in its title bar. This string should be the name of the client document containing the object. The client does not need to call **OleSetHostNames** every time an embedded object is activated, because the library maintains a record of the specified names.

6.3.9 Deleting Objects

To permanently delete an object from a document, the client should call the **OleDelete** function. **OleDelete** closes the specified object, if necessary, before deleting it.

6.3.10 Client Cut and Copy Commands

A client application can copy an object to the clipboard by simply opening the clipboard, calling the **OleCopyToClipboard** function, and closing the clipboard again. If the client supports delayed rendering, however, it should follow these steps to cut or copy an object to the clipboard:

1. Open and empty the clipboard.
2. Put the preferred data formats on the clipboard.
3. Call the **OleEnumFormats** function to retrieve the formats for the object.
4. Call the **SetClipboardData** function to put the formats on the clipboard, specifying NULL for the handle of the data.

   If the call to the **OleEnumFormats** function retrieves the ObjectLink format, the client should call **SetClipboardData** with OwnerLink instead of ObjectLink format. (For more information, see the following description of the **OleCopyToClipboard** function.)
5. Put any additional presentation data formats on the clipboard.
6. Close the clipboard.

To support the Cut command on the Edit menu, an application can call **OleCopyToClipboard** and then delete the object by using the **OleDelete** function. (The client can put only one of the selected objects on the clipboard, even when the user has selected and cut or copied multiple objects. In this case, the client typically puts the first object in the selection onto the clipboard.)

The **OleCopyToClipboard** function always copies OwnerLink format, not ObjectLink format, to the clipboard. For embedded objects, Native data always precedes the OwnerLink format. If a linked object uses Native data, OwnerLink format always precedes the Native data. If an application uses the **OleGetData** function to retrieve data from a linked object that has been copied by using **OleCopyToClipboard**, it should specify ObjectLink format, not OwnerLink format, even if OwnerLink format was put on the clipboard.

When an application that can act as both a client and server copies a selection to the clipboard that contains one or more objects, it should first allocate enough memory for the selection. To discover how much memory is required for each object, the application can call the **OleQuerySize** function. When memory has been allocated, the application should call the **OleRegisterClientDoc** function, specifying Clipboard for the document name. (In this case, the handle returned by the call to **OleRegisterClientDoc** identifies a document that is used only during the copy operation.) To save each object to memory, the application calls the **OleClone** function, calls the **OleSaveToStream** function for the cloned object, and then calls the **OleRelease** function to free the memory for the cloned object. When the selection has been saved to the stream, the application can call the **SetClipboardData** function. If **SetClipboardData** is successful, the application should call the **OleSavedClientDoc** function. The application then calls the **OleRevokeClientDoc** function, specifying the handle retrieved by the call to **OleRegisterClientDoc**.

For more information about the Cut and Copy commands, see Section 6.4.3, "Server Cut and Copy Commands."

6.3.11 Creating Objects

A client application can put linked and embedded objects in a document by pasting them from the clipboard, creating them from a file, copying them from other objects, or by starting a server application to create them directly.

6.3.11.1 Object-Creation Functions

Each of the following functions creates an embedded or linked object in a specified document:

Function	Description
OleClone	Creates an exact copy of an object.
OleCopyFromLink	Creates an embedded object that is a copy of a linked object.
OleCreate	Creates an embedded object of a specified class.
OleCreateFromClip	Creates an object from the clipboard. This function typically creates an embedded object.
OleCreateFromFile	Creates an object by using the contents of a file. This function typically creates an embedded object.
OleCreateFromTemplate	Creates an embedded object by using another object as a template.
OleCreateInvisible	Creates an object without displaying the server application to the user.
OleCreateLinkFromClip	Creates an object by using information on the clipboard. This function typically creates a linked object.

Function	Description
OleCreateLinkFromFile	Creates an object by using the contents of a file. This function typically creates a linked object.
OleObjectConvert	Creates an object that supports a specified protocol by converting an existing object.

Each of these functions requires a parameter that points to an **OLEOBJECT** structure when the function returns. Server applications often create an **OLEOBJECT** structure whenever an object is created; **OLEOBJECT** points to functions that describe how the server interacts with the object. Before the client library gives the client application a pointer to this structure, the library includes with the structure some internal information corresponding to the OwnerLink or ObjectLink data. This internal information allows the client library to identify the correct server when an OLE function such as **OleActivate** passes it a pointer to an **OLEOBJECT** structure. For more information about the **OLEOBJECT** structure, see Section 6.4.1, "Starting a Server Application."

Each new object must have a name that is unique to the client document. Although meaningful object names can be helpful, some applications assign unique object names simply by incrementing a counter for each new object. For more information about object names, see Section 6.3.3, "Document Management."

If a client application implements the Insert Object command, it should use the registration database to find out what OLE servers are available and then list those servers for the user. When the user selects one of the servers and chooses the OK button, the client can use the **OleCreate** function to create an object at the current position.

The **OleCopyFromLink**, **OleCreate**, and **OleCreateFromTemplate** functions always create an embedded object. The other object-creation functions can create either an embedded object or a linked object, depending on the order and type of available data.

If a client application's callback function receives the OLE_RELEASE notification after the client calls the **OleCreate** or **OleCreateFromFile** function, the client should respond by calling the **OleQueryReleaseError** function. If **OleQueryReleaseError** shows that there was an error when the object was created, the client application should delete the object.

Whenever an object-creation function returns OLE_WAIT_FOR_RELEASE, the calling application should either wait for the OLE_RELEASE notification or notify the user that the object cannot be created. For more information, see Section 6.3.6, "Asynchronous Operations."

If a client application accepts files dropped from File Manager, it should respond to the WM_DROPFILES message by calling the **OleCreateFromFile** function and specifying Packager for the lpszClass parameter.

6.3.11.2 Paste and Paste Link Commands

A client application should follow these steps to create an embedded or linked object by pasting from the clipboard:

1. Call the **OleQueryCreateFromClip** function to determine whether to enable the Paste command. If this function fails when StdFileEditing is specified for the *lpszProtocol* parameter, call it again, specifying Static.
2. Call the **OleQueryLinkFromClip** function to determine whether to enable the Paste Link command.
   - If the user chooses the Paste command, open the clipboard and call the **OleCreateFromClip** function.
   - If the user chooses Paste Link, open the clipboard and call the **OleCreateLinkFromClip** function.
3. Close the clipboard.
4. Call the **OleQueryType** function to determine the kind of object created by the creation function. (Depending on the order of clipboard data, **OleCreateFromClip** can sometimes create a linked object and **OleCreateLinkFromClip** can sometimes create an embedded object.)

The client application should put the pasted data or object into the document at the current position. The client should select the object so that the user can work with it immediately. If both the **OleQueryCreateFromClip** and **OleQueryLinkFromClip** functions fail but there is data on the clipboard that the client can interpret, the client should enable the Paste command.

If the information on the clipboard is incomplete—for example, if Native data is not accompanied by the OwnerLink format—the Paste command should insert a static object into the document. (A static object consists of the presentation data for an object; it cannot be edited by using standard OLE techniques. Attempts to open static objects fail and generate no notifications.)

If the client application implements the Paste Special command, it should use the **EnumClipboardFormats** function to produce a list of data formats on the clipboard. The client should also check the registration database to find the full name of the server application. The Paste Link button in the Paste Special dialog box works in exactly the same way as the Paste Link command on the Edit menu.

If the DDE Link format is available on the clipboard instead of ObjectLink format, the client application should perform the same link operation that it supported prior to the implementation of OLE.

6.3.12 Undo Command

A client application can use the **OleClone** function to support the Undo command. A cloned object is identical to the original except for connections to the server application; the cloned object is not automatically connected to the server. When the server is closed and the object is updated, the saved copy of the object gives the user the opportunity to undo all of the changes made in the server. Support for the Undo command is provided by the client application, because the server cannot maintain a record of the prior states of objects.

The Undo command restores an object to its condition prior to the last update from the server. To support this behavior, the client application must clone the object when it is first activated and then clone the updated object when an update occurs; the client must maintain two clones of the object. The clone of the original object must be maintained so that an updated object can be restored if the user chooses the Undo command. The clone of the updated object must be maintained to support the Undo command if the updated object is updated again. Because the data changes when the update occurs, the clone for supporting the Undo command must be made before any updates occur.

Because the client application cannot distinguish between different types of object activation, the client must clone an object for verbs that do not edit the object, even though no updates can occur in those cases.

6.3.13 Class Name Object Command

A client application can implement the *Class Name* Object command by using the **OleActivate** function. **OleActivate** includes a parameter that allows the client to specify the verb chosen by the user.

6.3.14 Links Command

When a user chooses the Links command, a dialog box appears listing every linked object in the document. The selected links are highlighted in the dialog box. The dialog box makes it possible for the user to invoke the verbs for an object, select whether link updating should be automatic or manual, update a link immediately, cancel a link, and repair broken links. For more information about this dialog box, see *Microsoft Windows User Interface Guidelines*.

The Links dialog box includes buttons that allow the user to activate the primary and secondary verbs for an object. A client application can implement these buttons by using the **OleActivate** function.

A client application can use the **OleGetLinkUpdateOptions** and **OleSetLinkUpdateOptions** functions to support the link-update radio buttons in the Links dialog box. The following are the three possible update options:

Option	Description
oleupdate_always	Update the linked object whenever possible. This option supports the Automatic link-update radio button in the Links dialog box.
oleupdate_onsave	Update the linked object when the source document is saved by the server.
oleupdate_oncall	Update the linked object only on request from the client application. This option supports the Manual link-update radio button in the Links dialog box.

These update options control when updates to the presentation of an object occur. The contents of the source document are used to update the presentation whenever the link is activated.

To support the Update Now button in the Links dialog box, an application can call the **OleUpdate** function. When a user chooses Update Now, the client application should update the links the user selected.

A user's choosing the Cancel Link button in the Links dialog box changes an object into a picture that an application cannot edit by using standard OLE techniques. An application can implement the Cancel Link button by using the **OleObjectConvert** function.

A client application should activate the Change Link button in the Links dialog box only if all the selected links are to the same source document. When the client has the correct information, it can repair the link by using the **OleGetData** and **OleSetData** functions. To retrieve the link information for an object, a client can call the **OleGetData** function, specifying the ObjectLink format. (The call to **OleGetData** fails if ObjectLink is specified and the object is not a link.) A client can retrieve class information by using **OleGetData** and specifying either the OwnerLink format (for embedded objects) or the ObjectLink format (for linked objects). The client can make it possible for the user to edit the link information and store it in the object by using the **OleSetData** function, specifying the ObjectLink format.

6.3.15 Closing a Client Application

A client application should use the **OleRelease** function to remove all objects from memory when it shuts down. If the library returns the value OLE_WAIT_FOR_RELEASE instead of OLE_OK, the client should not quit. The client can perform many cleanup tasks while waiting for the OLE_RELEASE notification—for example, it can close files, free memory, and hide windows.

The OLE_RELEASE notification to the client's callback function indicates that an operation has finished in a server application, but it does not identify the operation or indicate whether the operation was successful. A client application can call the **OleQueryReleaseStatus** function to determine whether an operation has been completed for a specified object. The **OleQueryReleaseMethod** function indicates the nature of the operation that has finished for a specified object. To discover the error value for the operation, the client can call the **OleQueryReleaseError** function.

If a client owns the clipboard when it quits, it should make sure that the data on the clipboard is complete and in the correct order.

6.4 Server Applications

An OLE server supplies functions that the server library calls when a user works with an object. The server library, OLESVR.DLL, uses DDE commands to communicate with the client library. When the client application calls one of the functions in the OLE API, the client library informs the server library and the server library routes the request to the appropriate function in the server-supplied list of function pointers.

In addition to the specialized functions that the server creates and which are called by the server library, there are ten OLE functions that allow a server to control the library's ability to gain access to the server and the documents and objects it controls:

Function	Description
OleBlockServer	Queues requests to the server until the server calls the **OleUnblockServer** function.
OleRegisterServer	Registers the specified server with the library. Information registered includes the class name and instance and whether the server supports single or multiple instances.
OleRegisterServerDoc	Registers a document with the server library.
OleRenameServerDoc	Renames the specified document.

Function	Description
OleRevertServerDoc	Restores a document to a previously saved state, without closing the document.
OleRevokeObject	Revokes access to the specified object.
OleRevokeServer	Revokes access to the specified server, closing any documents and ending communication with client applications.
OleRevokeServerDoc	Revokes access to the specified document.
OleSavedServerDoc	Informs the library that a document has been saved. Calling this function is equivalent to sending the OLE_SAVED notification.
OleUnblockServer	Processes a request from a queue created when the server application called the **OleBlockServer** function.

The **OleRevokeServer** and **OleRevokeServerDoc** functions can return OLE_WAIT_FOR_RELEASE. When a server application receives this error value, it should take the same action as a client application, dispatching messages until the server library calls the corresponding **Release** function.

6.4.1 Starting a Server Application

When a server application starts, it should follow these steps:

1. Register window classes and window procedures for the main window, documents, and objects.
2. Initialize the function tables for the **OLESERVERVTBL**, **OLESERVERDOCVTBL**, and **OLEOBJECTVTBL** structures.
3. Register the clipboard formats.
4. Allocate memory for the **OLESERVER** structure.
5. Register the server with the library by calling the **OleRegisterServer** function.
6. Check for the **/Embedding** and **/Embedding***filename* options on the command line and act according to the following guidelines. (Applications should also check for **-Embedding** whenever they check for these options.)
   - If neither **/Embedding** nor **/Embedding***filename* is present, call the **OleRegisterServerDoc** function, specifying an untitled document.
   - If the **/Embedding** option is present, do not register a document or display a window. (In this case, the server takes actions only in response to calls from the server library.)
   - If the **/Embedding***filename* option is present, do not display a window. Process the filename string and call the **OleRegisterServerDoc** function.

The **OLESERVERVTBL**, **OLESERVERDOCVTBL**, and **OLEOBJECTVTBL** structures are tables of function pointers. The server library uses these structures to route requests from the client application to the server. The server application should create the function pointers in these structures by using the **MakeProcInstance** function. The functions should also be exported in the application's module-definition file.

The **OLESERVER** structure contains a pointer to an **OLESERVERVTBL** structure. The **OLESERVERVTBL** structure contains pointers to functions that control such fundamental server tasks as opening files, creating objects, and terminating after an editing session. Several of the functions pointed to by the **OLESERVERVTBL** structure cause the server to allocate and initialize an **OLESERVERDOC** structure.

The **OLESERVERDOC** structure contains a pointer to an **OLESERVERDOCVTBL** structure. The **OLESERVERDOCVTBL** structure contains pointers to functions that control such tasks as saving or closing documents or setting document dimensions. The **OLESERVERDOCVTBL** structure also contains a function that causes the server to allocate and initialize an **OLEOBJECT** structure.

The **OLEOBJECT** structure contains a pointer to an **OLEOBJECTVTBL** structure. The **OLEOBJECTVTBL** structure contains pointers to functions that operate on objects. After the server application creates an **OLEOBJECT** structure, the server library gives information about the structure to the client library. The client library then creates a parallel **OLEOBJECT** structure (including internal information identifying the server application, the document, and the item for the object) and passes a pointer to that structure to the client application.

This hierarchy of structures—**OLESERVER**, **OLESERVERDOC**, and **OLEOBJECT**—makes it possible for a server to open as many documents as the library requests and for each document to contain as many objects as necessary.

A server application can register the clipboard formats by calling the **RegisterClipboardFormat** function for each format, specifying Native, OwnerLink, ObjectLink, and any other formats it requires.

When the server application starts, it creates an **OLESERVER** structure and then registers it with the library by calling the **OleRegisterServer** function. When this function returns, one of its parameters points to a server handle. The library uses this handle of refer to the server, and the server uses it in calls to the server-specific OLE functions.

If an OLE server application is also a DDE server, the class name specified in the call to the **OleRegisterServer** function cannot be the same as the name of the executable file for the application.

When a client working with a compound document opens a linked or embedded object for editing, the client library starts the server using the **/Embedding** command-line option. The server uses this option to determine whether the object has been opened directly by a user or as part of an editing session for linked and embedded objects. (If the object is a linked object, the **/Embedding** option is followed by a filename.) When a server is started for an embedded object with the **/Embedding** option, the server should not create a document or show a window. Instead, it should call the **OleRegisterServer** function and then enter a message-dispatch loop. (If the server is started with the **/Embedding***filename* option, it should also call the **OleRegisterServerDoc** function.) The server then takes actions in response to calls from the library. The server should not make itself visible until the library calls the **Show** or **DoVerb** function in the **OLEOBJECTVTBL** structure. (Server applications should check for both **–Embedding** and **/Embedding**.)

By calling the **OleBlockServer** function, a server application can cause requests from the client library to be saved in a queue. When the server is ready for the server library to process the requests, it can call the **OleUnblockServer** function. It is best to use the **OleUnblockServer** function prior to the **GetMessage** function in a message loop, so that all blocked requests are unblocked before getting the next message. (Often a server returns OLE_BUSY instead of calling **OleBlockServer**. Returning OLE_BUSY has two advantages: It allows the client to decide whether to retry the message or discontinue the operation, and it allows the server to choose which requests to process.)

When an error occurs in a server-supplied function, the server should return the **OLESTATUS** error value that best describes the error. The OLE libraries use these error values to help determine the appropriate behavior in error situations. However, the client application does not necessarily receive the error values the server returns; the OLE libraries may change error values before passing them to the client application.

6.4.2 Opening a Document or Object

Whenever the server library calls the **Open**, **Create**, **CreateFromTemplate**, or **Edit** function in the **OLESERVERVTBL** structure, the server creates an **OLESERVERDOC** structure. If the document is opened by a call from the server library, the server application returns the **OLESERVERDOC** structure to the library. If the document is opened directly by a user, however, the server should call the **OleRegisterServerDoc** function to register the document with the library. The library then uses the **GetObject** function in the **OLESERVERDOCVTBL** structure to request the server to create an **OLEOBJECT** structure for each object requested by the client application.

A new instance of the server application is typically started when the client activates a linked or embedded object. This new instance is unnecessary if the object is already open in an instance of the server or if the server is a single-instance (MDI) server that is already open. For more information about the rules for starting new instances of server applications, see *Microsoft Windows User Interface Guidelines*.

Whether the server library starts a new instance of a server to edit an embedded or linked object depends upon the value specified when the server calls the **OleRegisterServer** function.

6.4.3 Server Cut and Copy Commands

A server application should follow these steps to cut or copy onto the clipboard data that a client can then use to create an embedded or linked object:

1. Open and empty the clipboard.
2. Put the data formats that describe the selection on the clipboard, using the **SetClipboardData** function.
3. Close the clipboard.

If the server cuts data onto the clipboard, rather than copying it, the server typically does not offer ObjectLink or Link formats, because the source for the data has been removed from the document.

The server should put data on the clipboard in the order given in Section 6.2.5, "Clipboard Conventions."

Typically, the server puts server-specific formats, Native format, OwnerLink format, and presentation formats on the clipboard. If it can support links, the server also puts ObjectLink format and, when appropriate, Link format on the clipboard. The server must provide a presentation format (CF_METAFILE, CF_BITMAP, or CF_DIB) if the server does not have an object handler. Native data can be used as a presentation format only if the server has an object handler that can use the Native data.

If a user copies onto the clipboard a selection that includes an embedded object or a link, the data formats the server should copy depend upon whether the container document modifies the object or link. If the document does not modify the object or link, the best formats are the Native and OwnerLink formats from the original source of the object. If the document modifies the object or link—for example, by recoloring it—the best formats are the Native and OwnerLink formats from the container document.

If a server uses a metafile as the presentation format for an object, the mapping mode for that metafile must be MM_ANISOTROPIC. When a server application uses fonts in these metafiles, it can improve performance by using TrueType fonts. (Metafiles scale better when they use TrueType fonts.) To use TrueType fonts exclusively, the server should set bit 2 (04h) of the **lpPitchAndFamily** member of the **LOGFONT** structure.

The OLE libraries express the size of every object in MM_HIMETRIC units. Neither the width nor height of an object should exceed 32,767 MM_HIMETRIC units.

6.4.4 Update, Save As, and New Commands

When a server is started as part of editing an object from within a compound document, the server application should change the Save command on the File menu to Update. When the user chooses the Update command, the server should call the **OleSavedServerDoc** function.

When the user chooses the Save As, New, or Open command in a single-document server, the application should display a message asking the user whether to update the object in the compound document before performing the action. When the user chooses the Save As command, the server should call the **OleRenameServerDoc** function. If the user responds to the message by choosing to save changes in the object before renaming the document, the server should call the **OleSavedServerDoc** function before calling **OleRenameServerDoc**. For embedded objects, choosing the Save As command causes the connection with the client to be broken, because this command reassociates a document in memory with the specified new file. For linked objects, calling **OleRenameServerDoc** when the user chooses Save As makes it possible for the client to associate the link with the new file.

Most server applications maintain a "dirty" flag that records whether changes have been made to each open document in an instance. The following table shows the rules that apply to this flag when the server edits an embedded object. By following these rules, a server can ensure that this flag is TRUE when the document being edited in the server matches the embedded object in the client and that, otherwise, this flag is FALSE.

Flag	Condition
TRUE	Library calls the **Create** function in the **OLESERVERVTBL** structure.
TRUE	Library calls the **CreateFromTemplate** function in **OLESERVERVTBL**.
TRUE	Document is changed in server.
FALSE	Library calls the **Edit** function in **OLESERVERVTBL**.
FALSE	Library calls the **GetData** function in **OLEOBJECTVTBL** with the Native data format. (The flag should not change for any other formats.)

A server following these rules displays the message asking whether to update the object whenever it destroys a document that was editing an embedded object and the "dirty" flag is TRUE.

In an MDI server application, the New and Open commands on the File menu simply open a new window, and the connection with the client application remains unchanged. The user can continue to work with the server application after choosing one of these commands, but when the user exits the server application, the focus does not necessarily return to the client application.

Typically, a server can call the **OleSavedServerDoc** function whenever an object needs to be updated in the client document, including when the server closes the document. When the server closes the document and the object should be updated, the server sends the OLE_CLOSED notification. Client applications receive the OLE_CLOSED notification for embedded objects but not for linked objects, because the server library intercepts the notification for linked objects.

6.4.5 Closing a Server Application

The server library calls the **Exit** function in the **OLESERVERVTBL** structure when the server must quit. The server library calls the **Release** function to inform the server that it is safe to quit; the server does not necessarily stop when the library calls **Release**.

The server must exit when it is invisible and the library calls **Release**. (The only exception is when an application supports multiple servers; in this case, an invisible server is sometimes not revocable when the library calls **Release**.) If the server has no open documents and it was started with the **/Embedding** option (indicating that it was started by a client application), the server should exit when the library calls the **Release** function. If the user explicitly loads a document into a single-instance (MDI) server, however, the server should not exit when the library calls **Release**.

When the user closes a server that has edited an embedded object without updating changes to the client application, the server should display a message asking whether to save the changes. If the user chooses to save the changes, the server should send the OLE_CLOSED notification and call the **OleRevokeServerDoc** function. (Because sending OLE_CLOSED prompts the server library to send data to the client library, it is not necessary to send OLE_CHANGED or OLE_SAVED. If the user chooses not to save the changes, the server should simply call the **OleRevokeServerDoc** function (without sending OLE_CLOSED).

A server can use the **OleRevokeObject** function to revoke a client's access to an object—for example, if the user destroys the object. Similarly, the **OleRevoke-ServerDoc** function revokes a client's access to a document. (Because **Ole-RevokeServerDoc** revokes a client's access to all objects in a document, an application that uses **OleRevokeServerDoc** does not need to call the **OleRevoke-Object** function for objects in that document.) To terminate all conversations with client applications, the server can call the **OleRevokeServer** function. These functions inform the server library that the specified items are no longer available.

A server application can receive OLE_WAIT_FOR_RELEASE—for example, the **OleRevokeServerDoc** function can return this value. Although a server can enter a message-dispatch loop and wait for the library to call the server's **Release** function, servers should never enter message-dispatch loops inside any of the server-supplied functions that are called by the server library.

The client application should not instruct the server to close the document or exit when the server is editing a linked object, unless the server is updating the link without displaying the object to the user. Because a linked object exists independently of the client, the user controls saving and closing the document by using the server application.

If a server application owns the clipboard when it closes, it should make sure that the data on the clipboard is complete and in the correct order. For example, any Native data should be accompanied by the OwnerLink format.

6.5 Object Handlers

An application developer can use object handlers to introduce customized features into implementations of linked and embedded objects. When an object handler exists for a class of object, the object handler supplants some or all of the functionality that is usually provided by the client library and the server application. The object handler can take specialized action for any of the functions it intercepts. The object handler passes functions that it does not take action on to the client library, which then implements the default processing for that class.

An application might use an object handler to render Native data as the presentation data for an object, instead of using metafiles or bitmaps. Object handlers could also be used to implement special behavior when an object is opened.

6.5.1 Implementing Object Handlers

A server installing an object handler registers the handler with the registration database, using the keyword **handler**. Whenever a client application calls one of the object-creation functions, the client library uses the class name specified for the object and the **handler** keyword to search the registration database. If the library finds an object handler, the client library loads the handler and calls it to create the object. The handler can create an object for which all of the creation functions and methods are defined by the handler, or it can call default object-creation functions in the client library.

The client library exports the object-creation OLE functions with new names; in each case, the prefix "Ole" is changed to "Def" (for "default"). Object handlers can import any of these functions and use them when creating objects.

Object handlers must import the following functions:

OLE function	Name exported by client library
OleCreate	DefCreate
OleCreateFromClip	DefCreateFromClip
OleCreateFromFile	DefCreateFromFile
OleCreateFromTemplate	DefCreateFromTemplate
OleCreateLinkFromClip	DefCreateLinkFromClip
OleCreateLinkFromFile	DefCreateLinkFromFile
OleLoadFromStream	DefLoadFromStream

When an object handler defines a function that is to be called by the client application, it should use the same name as the corresponding OLE function the client calls, with the prefix "Ole" replaced by "Dll". For example, when an object handler uses the **DefCreate** function exported by the client library, the handler should use it inside a function named **DllCreate**. When the client library finds an object handler for a class of object, it calls handler-specific object-creation functions by specifying this "Dll" prefix.

When the handler calls one of the default object-creation functions, it receives a handle of an **OLEOBJECT** structure, which in turn points to the **OLEOBJECTVTBL** structure containing the current object-management functions. The object handler should copy this **OLEOBJECTVTBL** structure and customize the structure by replacing any function pointers in the structure with pointers to functions of its own. (If the object handler saves the pointers to the default functions, any of the replacement functions can also call the default functions in the table of function pointers.) When the object handler has finished customizing the structure, it should replace the pointer to the old **OLEOBJECTVTBL** structure with a pointer to the modified **OLEOBJECTVTBL** structure.

When the client makes a call to a function in the client library, the call is dispatched through the object handler's **OLEOBJECTVTBL** structure. If the object handler has replaced the function pointer, the call is routed to the function supplied by the handler. Otherwise, the call is routed to the client library.

Each **OLECLIENT**, **OLEOBJECT**, **OLESERVER**, **OLESERVERDOC**, or **OLESTREAM** structure contains a pointer to a structure that contains a table of function pointers. (Structures containing tables of function pointers are identified with the "VTBL" suffix.) Each of the structures containing a pointer to a "VTBL" structure can also contain extra instance-specific information. This information is meaningful only to the application that supplies it and should not be used by other applications; for example, an object handler should not attempt to use any instance-specific information in an **OLECLIENT** structure.

The object handler should use the "Def" and "Dll" renaming conventions when it defines specialized functions. For example, if an object handler modifies the **Draw** function from an object's **OLEOBJECTVTBL** structure, it should copy that **Draw** function to a function named **DefDraw** and replace the **Draw** function with a specialized function named **DllDraw**. Inside the **DllDraw** function, the object handler can call **DefDraw** if the default drawing operation is appropriate in a particular case.

The following example demonstrates this process of copying and replacing pointers to functions. Functions with the "Dll" prefix should be exported in the module-definition file.

```
/* Declare the DllDraw and DefDraw functions.           */

OLESTATUS FAR PASCAL DllDraw(LPOLEOBJECT, HDC, LPRECT, LPRECT, HDC);
OLESTATUS (FAR PASCAL *DefDraw)(LPOLEOBJECT, HDC, LPRECT, LPRECT, HDC);

/* Copy the Draw function from OLEOBJECTVTBL to DefDraw. */

    DefDraw  = lpobj->lpvtbl->Draw;

/* Copy DllDraw to OLEOBJECTVTBL.                        */

    *lpobj->lpvtbl->Draw = DllDraw;


OLESTATUS FAR PASCAL DllDraw(lpObject, hdc, lpBounds, lpWBounds,
        hdcFormat)
LPOLEOBJECT     lpObject;
HDC             hdc;
LPRECT          lpBounds;
LPRECT          lpWBounds;
HDC             hdcFormat;
{
    /* Return DefDraw if Native data is not available.   */

    if ((*lpobj->lpvtbl->GetData) (lpobj, cfNative, &hData) != OLE_OK)
        return (*DefDraw) (lpobj, hdc, lpBounds, lpWBounds, hdcFormat);
    .
    .
    .
}
```

6.5.2 Creating Objects in an Object Handler

Most of the object-creation functions in the OLE API work in exactly the same way when they are renamed and used by object-handler DLLs. Two functions are somewhat different, however: **OleCreateFromClip** and **OleLoadFromStream**.

6.5.2.1 DefCreateFromClip and DllCreateFromClip

When the client library calls the **DllCreateFromClip** function, the library includes a parameter that is not specified in the original call to the **OleCreate-FromClip** function. This parameter, *objtype*, specifies whether the object being created is an embedded object or a link; its value can be either OT_LINK or OT_EMBEDDED.

The following syntax block shows the *objtype* parameter when an object handler uses the **DefCreateFromClip** function. The **DllCreateFromClip** function has exactly the same syntax as **DefCreateFromClip**. For a full description of all the parameters, see the description of the **OleCreateFromClip** function in the *Microsoft Windows Programmer's Reference, Volume 2.*

```
OLESTATUS DefCreateFromClip(lpszProtocol, lpclient, lhclientdoc,
    lpszObjname, lplpobject, renderopt, cfFormat, objtype);
LPSTR lpszProtocol;           /* address of string for protocol name */
LPOLECLIENT lpclient;         /* address of client structure         */
LHCLIENTDOC lhclientdoc;      /* long handle of client document      */
LPSTR lpszObjname;            /* string for object name              */
LPOLEOBJECT FAR * lplpobject; /* address of pointer to object        */
OLEOPT_RENDER renderopt;      /* rendering options                   */
OLECLIPFORMAT cfFormat;       /* clipboard format                    */
LONG objtype;                 /* OT_LINKED or OT_EMBEDDED            */
```

If **DllCreateFromClip** calls **DefCreateFromClip**, **DllCreateFromClip** should pass it the *objtype* parameter along with the other parameters from the version of **DefCreateFromClip** that was exported by the client library. **DllCreateFromClip** can modify some of these parameters before passing them back to **DefCreate-FromClip**. For example, the object handler could specify a different value for the *renderopt* parameter when it calls **DefCreateFromClip**. If the client calls this function with **olerender_draw** for *renderopt* and the handler performs the drawing with Native data, the handler could change **olerender_draw** to **olerender_none**. If the client calls this function with **olerender_draw** for *renderopt* and the handler calls the **GetData** function and performs the drawing based on a class-specific format, the handler could change **olerender_draw** to

olerender_format. If the handler needed a different rendering format than the format specified by the client application, the object handler could also change the value of the *cfFormat* parameter in the call to **DefCreateFromClip**.

If an object handler uses Native data to render an embedded object, the handler can call the library and specify **olerender_none**. If a handler uses Native data to render a linked object, it can use **olerender_format** and specify Native data. When the handler's **Draw** function is called, the handler calls the **GetData** function, specifying Native data, to do the rendering. If a handler uses a private data format, the procedure is the same—except that the private format is specified with the **olerender_format** option and with the **GetData** function.

6.5.2.2 DefLoadFromStream and DllLoadFromStream

When the client library calls the **DllLoadFromStream** function, the library includes three parameters that are not specified in the original call to the **OleLoadFromStream** function. One of the additional parameters is *objtype*, as described for **DefCreateFromClip** and **DllCreateFromClip**. The other two parameters are *aClass*, which is an atom containing the class name for the object, and *cfFormat*, which specifies a private clipboard format that the object handler can use for rendering the object.

The following syntax block shows the *objtype*, *aClass*, and *cfFormat* parameters when an object handler uses the **DefLoadFromStream** function. The **DllLoadFromStream** function has exactly the same syntax as **DefLoadFromStream**. For a full description of all the parameters, see the description of the **OleLoadFromStream** function in the *Microsoft Windows Programmer's Reference*, *Volume 2*.

```
OLESTATUS DefLoadFromStream(lpstream, lpszProtocol, lpclient,
    lhclientdoc, lpszObjname, lplpobject, objtype, aClass, cfFormat);
LPOLESTREAM lpstream;          /* address of stream for object       */
LPSTR lpszProtocol;            /* address of string for protocol name */
LPOLECLIENT lpclient;          /* address of client structure        */
LHCLIENTDOC lhclientdoc;       /* long handle of client document     */
LPSTR lpszObjname;             /* string for object name             */
LPOLEOBJECT FAR * lplpobject;  /* address of pointer to object       */
LONG objtype;                  /* OT_LINKED or OT_EMBEDDED           */
ATOM aClass;                   /* atom containing object's class name */
OLECLIPFORMAT cfFormat;        /* private data format for rendering  */
```

If **DllLoadFromStream** calls **DefLoadFromStream**, **DllLoadFromStream** should pass it the three additional parameters along with the other parameters from the version of **DefLoadFromStream** that was exported by the client library.

DllLoadFromStream can modify some of these parameters before passing them back to **DefLoadFromStream**. For example, the object handler could modify the value of the *cfFormat* parameter to specify a private data format it would use to render the object.

When the client calls the object handler with **DefLoadFromStream**, the handler uses the **Get** function from the **OLESTREAMVTBL** structure to obtain the data for the object.

6.6 Direct Use of Dynamic Data Exchange

The OLE libraries, OLECLI.DLL and OLESVR.DLL, use DDE messages to communicate with each other. Although client and server applications can use DDE directly, without employing OLECLI.DLL or OLESVR.DLL, this method of implementing OLE is not recommended. Future enhancements to the OLE libraries will benefit applications that use the libraries but will not benefit applications that use DDE directly.

The following information about the DDE-based OLE protocol is provided for applications that must implement DDE directly, despite losing the ability to take advantage of future enhancements to the system.

Implementation of the OLE protocol requires implementation of the underlying DDE protocol. All the standard DDE rules and facilities apply. Applications that conform to this protocol must also conform to the DDE specification. Conforming to this specification implies supporting the System topic and the standard items in that topic.

6.6.1 Client Applications and Direct Use of Dynamic Data Exchange

When opening a link or an embedded document, the client application should look up the class name in the registration database, as described in Section 6.2.6, "Registration."

The following pseudocode illustrates the chain of events for a client implementing OLE through DDE. Whenever a client that attempts to establish a conversation with a server receives responses from more than one server, the client should accept the first server and reject the others.

Linked object:

```
WM_DDE_INITIATE class name, document name
if not found {
    WM_DDE_INITIATE class name, OLESystem
    if not found {
        WM_DDE_INITIATE class name, System
        if not found {
            launch application name, /Embedding
            fLaunched = true
            WM_DDE_INITIATE class name, OLESystem
            if not found {
                WM_DDE_INITIATE class name, System
                if not found
                    return error
            }
        }
    }

    /*
     * Now there is a conversation with the server on the System or
     * OLESystem topic.
     */

    WM_DDE_EXECUTE StdOpenDocument(DocumentName)
    WM_DDE_INITIATE class name, document name
    if not found {
        if(fLaunched) WM_DDE_EXECUTE StdExit /* clean up */
            return error
    }
}

/*
 * Now there is a conversation with the correct document.
 */
```

Embedded object:

```
WM_DDE_INITIATE class name, OLESystem
if not found {
    WM_DDE_INITIATE class name, System
    if not found {
        launch application name, /Embedding
        fLaunched = true
        WM_DDE_INITIATE class name, OLESystem
        if not found {
            WM_DDE_INITIATE class name, System
            if not found
                return error
        }
    }
}

/*
 * Now there is a conversation with the server on the system or
 * OLESystem topic.
 */

DDE_EXECUTE StdEditDocument(DocumentName)

/*
 * Or StdCreateDoc if this is an Insert Object command
 */

WM_DDE_INITIATE class name, document name
if not found {
    if(fLaunched) DDE_EXECUTE StdExit        /* clean up */
        return error
}

/* Now there is a conversation with the correct document. */
```

6.6.2 Server Applications and Direct Use of Dynamic Data Exchange

When a server receives the **/Embedding** command-line argument, it should not create a new default document. Instead, it should wait until the client sends either the **StdOpenDocument** command or the **StdEditDocument** command followed by the Native data and then instructs the server to show the window. The server can use the **StdHostNames** item to display the client's name in the window title.

The following pseudocode illustrates the chain of events for a server implementing OLE through DDE. The example shows two cases: one in which the server reuses a single instance for editing all objects (in MDI child windows), and another in which a new instance is used for each object. Applications that use a new instance for each object should reject requests to open or create a new document when they already have a document open.

MDI application:

```
case WM_DDE_INITIATE:
  if class name == this class {
    if (DocumentName == OLESystem || DocumentName == System)
        WM_DDE_ACK
    else if DocumentName == name of some open document
        WM_DDE_ACK
  }
```

Multiple-instance application:

```
case WM_DDE_INITIATE:
  if class name == this class {
    if (DocumentName == OLESystem || DocumentName == System) {
        if no documents are open
            WM_DDE_ACK
    }
    else if DocumentName == name of some open document
        WM_DDE_ACK
  }
```

6.6.3 Conversations

Document operations are performed during conversations with an application's OLESystem or System topic. The document's class name is used to establish the conversation.

Data transfer and negotiation operations are performed during conversations with the document (that is, the topic). The document name is used to establish the conversation.

Note that the topic name is used only in initiating conversations and is not fixed throughout the conversation; permitting the document to be renamed does not mean that there will be two names. Therefore, it is reasonable to tie the topic name to the document name.

6.6.4 Items for the System Topic

An application using DDE-based OLE can use three new items for the System topic: the Topics item, the Protocols item, and the Status item.

The Topics item returns a list of DDE topic names that the server application has open. Where topics correspond to documents, the topic name is the document name.

The Protocols item returns a list of protocol names supported by the application. The list is returned in tab-separated text format. A protocol is a defined set of DDE execute strings and item and format conventions that the application understands. The protocol currently defined for linked and embedded objects is the following:

Protocol: StdFileEditing *commands/items/formats*

For compatibility with client applications that were written before the implementation of the OLE protocol, server applications that use the DDE protocol directly should also include the string Embedding in the list of protocols.

The Status item is a text item that returns Ready if the server is prepared to respond to DDE requests; otherwise, it returns Busy. This item can be queried to determine if the client should offer such functions as one that gives the user an opportunity to update the object. Because it is possible that a server could reject or defer a request even if Status returns Ready, client applications should not depend solely on the Ready item.

6.6.5 Standard Item Names and Notification Control

Applications supporting OLE with direct DDE use four clipboard formats in addition to the regular data and picture formats. These are ObjectLink, OwnerLink, Native, and Binary. Binary format is a stream of bytes whose interpretation is implicit in the item; for example, the **EditEnvItems**, **StdTargetDevice**, and **StdHostNames** items are in Binary format. The ObjectLink, OwnerLink, and Native formats are described in Section 6.2.5, "Clipboard Conventions."

New items available on each topic other than the System topic are defined for this protocol. These items are the following:

Item	Description
StdDocumentName	Contains the permanent document name associated with the topic. If no permanent storage is associated with the topic, this item is empty. This item supports both request and advise transactions and can be used to detect the renaming of open documents.
EditEnvItems	Returns a list in tab-separated text format of the items that contain environmental information supported by the server for its documents. Currently defined items are **StdHostNames**, **StdDocDimensions**, and **StdTargetDevice**. Applications can declare other items (and define their interpretations if Binary format is used) to permit clients that are informed of these items to provide more detailed information. Servers that cannot use particular items should omit their names from the EditEnvItems item. Clients should use the WM_DDE_REQUEST message with this item to find out which items the server can use and should supply the data through a WM_DDE_POKE message.
StdHostNames	Accepts information about the client application, in Binary format interpreted as the following structure: `struct { WORD clientNameOffset; WORD documentNameOffset; BYTE data[]; } StdHostNames;` The offsets are relative to the start of the data array. They indicate the starting point for the appropriate information in the array.
StdTargetDevice	Accepts information about the target device that the client is using. This information is in Binary format, interpreted as the following structure. Offsets are relative to the start of the data array. `typedef struct _OLETARGETDEVICE { WORD otdDeviceNameOffset; WORD otdDriverNameOffset; WORD otdPortNameOffset; WORD otdExtDevmodeOffset; WORD otdExtDevmodeSize; WORD otdEnvironmentOffset; WORD otdEnvironmentSize; BYTE otdData[]; } OLETARGETDEVICE;`

```
struct {
    WORD clientNameOffset;
    WORD documentNameOffset;
    BYTE data[];
} StdHostNames;
```

```
typedef struct _OLETARGETDEVICE {
    WORD otdDeviceNameOffset;
    WORD otdDriverNameOffset;
    WORD otdPortNameOffset;
    WORD otdExtDevmodeOffset;
    WORD otdExtDevmodeSize;
    WORD otdEnvironmentOffset;
    WORD otdEnvironmentSize;
    BYTE otdData[];
} OLETARGETDEVICE;
```

Item	Description
StdDocDimensions	Accepts information about the size of a document. This information is in Binary format, interpreted as the following structure. These values are specified in MM_HIMETRIC units. `struct { int iXContainer; int iYContainer; } StdDocDimensions;`
StdColorScheme	Returns the colors that the server is currently using and accepts information about the colors that the client requests the server to use. This information is in Binary format, interpreted as a **LOGPALETTE** structure.
null	Specifies a request or advise transaction on all data contained in the topic. This item is a zero-length item name.

The update method used for advise transactions on items follows a convention in which an update specifier is appended to the actual item name. The item is encoded as follows:

itemname/update type

For backward compatibility, omitting the update type has the same result as specifying **/Change**. The *update type* placeholder may be filled with one of the following values:

Value	Meaning
/Change	Notify for each change.
/Close	Notify when document is closed.
/Save	Notify when document is saved.

DDE server applications are required to save each occurrence of a WM_DDE_ADVISE message that specifies a unique combination of *itemname, update type, format,* and *conversation.* A notification is disabled by a WM_DDE_UNADVISE message with corresponding parameters. If the WM_DDE_UNADVISE message does not specify a format, it disables the oldest notification in first in, first out (FIFO) rotation.

6.6.6 Standard Commands in DDE Execute Strings

The syntax for standard commands sent in execute strings is the same as for other DDE commands:

command(argument1,argument2,...)[command2(argument1,argument2,...)]

Commands without arguments do not require parentheses. String arguments must be enclosed in double quotes.

6.6.6.1 International Execute Commands

DDE execute strings are typically sent from a macro language in an external application and are typically localized. OLE execute commands, however, are sent by application programs for their own purposes, need not be localized, and must be commonly recognized.

The OLE standard execute commands should not be localized; the U.S. spelling and separator characters are used. Therefore, the following rules apply:

- Client applications and the client library send standard execute commands in U.S. form.
- The server library must receive the U.S. form for these commands.
- Servers written directly to the DDE-level protocol should parse the U.S. form, if they have no additional commands.
- Servers that support both OLE and localized DDE execute commands should first parse the string by using localized separators. If this fails, they should parse it again using the U.S. form and, if successful, should execute the command. Optionally, if the command is received in the U.S. form, the server can check that the command is one of the valid standard commands.

6.6.6.2 Required Commands

This section lists commands that must be supported by server applications.

The **StdNewDocument**, **StdNewFromTemplate**, **StdEditDocument**, and **StdOpenDocument** commands all make the document available for DDE conversations with the name *DocumentName*. They do not show any window associated with the document; the client must send the **StdShowItem** and **StdDoVerbItem** commands, or the **StdDoVerbItem** command alone to make the window visible. This enables the client to negotiate additional parameters with the server (for example, the **StdTargetDevice** item) without causing unnecessary repaints.

StdNewDocument(*ClassName*, *DocumentName***)**

Creates a new, empty document of the given class, with the given name, but does not save it. The server should return an error value if the document name is already in use. When the client receives this error, it should generate another name and try again.

The server should not show the window until it receives a **StdShowItem** command. Waiting for the client to send the **StdShowItem** and **StdDoVerbItem** commands makes it possible for the client to negotiate additional parameters (for example, by using **StdTargetDevice**) without forcing the window to repaint.

StdNewFromTemplate(*ClassName***,** *DocumentName***,** *TemplateName***)**
Creates a new document of the given class with the given document name, using the template with the given permanent name (that is, filename).

The server should not show the window until it receives a **StdShowItem** command. Waiting for the client to send a **StdShowItem** command makes it possible for the client to negotiate additional parameters (for example, by using **StdTargetDevice**) without forcing the window to repaint.

StdEditDocument(*DocumentName***)**
Creates a document with the given name and prepares to accept data that is poked into it with WM_DDE_POKE. The server should return an error if the document name is already in use. When the client receives this error, it should generate another name and try again.

The server should not show the window until it receives a **StdShowItem** command. Waiting for the client to send a **StdShowItem** command makes it possible for the client to negotiate additional parameters (for example, by using **StdTargetDevice**) without forcing the window to repaint.

StdOpenDocument(*DocumentName***)**
Sent to the System topic. This command opens an existing document with the given name.

The server should not show the window until it receives a **StdShowItem** command. Waiting for the client to send a **StdShowItem** command makes it possible for the client to negotiate additional parameters (for example, by using **StdTargetDevice**) without forcing the window to repaint.

StdCloseDocument(*DocumentName***)**
Sent to the System topic. This command closes the window associated with the document. Following acknowledgment, the server terminates any conversations associated with the document. The server should not activate the window while closing it.

StdShowItem(*DocumentName***,** *ItemName* **[,***fDoNotTakeFocus***])**
Sent to the System topic. This command makes the window containing the named document visible and scrolls to show the named item (if any). The optional third argument indicates whether the server should take the focus and bring itself to the front. This argument should be TRUE if the server should not take the focus; otherwise, it should be FALSE. The default value is FALSE.

StdExit
Shuts down the server application. This command should be used only by the client application that launched the server. This command is available in the System topic only.

StdExit is sent to shut down an application if an error occurs during the startup phase or if the client started the server for an invisible update. If servers have unsaved data opened by the user, they should ignore this command.

6.6.6.3 Variants on Required Commands

The following variants of the above commands may be sent to the document topic rather than the System topic. This allows a client that already has a conversation with the document to avoid opening an additional conversation with the system. The document name is omitted from these commands because it is implied by the conversation topic and because it may have been changed by the server. This kind of name change does not invalidate the conversation. The client should not be forced to keep track of the name change unnecessarily. However, the server must be able to use the conversation information to identify the document on which to operate.

StdCloseDocument
Sent to the document conversation. This command closes the document associated with the conversation without activating it. This command causes a WM_DDE_TERMINATE message to be posted by the server window following the acknowledgment.

StdDoVerbItem(*ItemName, iVerb, fShow, fDoNotTakeFocus*)
Sent to the document conversation. This command is similar to the **StdShowItem** command, except that it includes an integer indicating which of the registered operations to perform and a flag indicating whether to show the window. The server can ignore the *fShow* flag, if necessary.

StdShowItem(*ItemName* [, *fDoNotTakeFocus*])
Sent to the document conversation. This command shows the document window, scrolling if necessary to bring the item into view. If the item name is NULL, scrolling does not occur. The optional second argument indicates whether the server should take the focus and bring itself to the front. This argument should be TRUE if the server should not take the focus; otherwise, it should be FALSE. The default value is FALSE.

Shell Library

Chapter 7

This chapter describes features of the shell for the Microsoft Windows operating system. The following features are supported by the dynamic-link library SHELL.DLL:

- The registration database
- The drag-drop feature
- Using associations to find and start applications
- Extracting icons from executable files

7.1 Registration Database

The registration database is a systemwide source of information about applications. This information is used to support the integration of applications with Windows File Manager and is used by applications that support object linking and embedding (OLE).

An application can use the registration database to store the following information:

- The name of the executable file that is associated with a given filename extension
- The command line to execute—or dynamic data exchange (DDE) messages to send—when the user opens a file from Windows shell applications (File Manager or Program Manager)
- The command line to execute—or DDE messages to send—when the user prints a file from File Manager
- Details about the implementation of OLE if the application is an OLE server

The registration database is a standard part of Windows version 3.1. Any Windows version 3.0 application that supports OLE also uses the registration database. The registration database is not meant as a place for applications to store private data. Applications should use private initialization files for data that is not defined or that is not needed either by the Windows 3.1 shell applications or by OLE applications.

For most applications, the developer uses Microsoft Windows Registration Editor (REGEDIT.EXE) to edit the registration database and produce a registration (.REG) file that contains readable text strings corresponding to database entries. This .REG file can be merged into the user's registration database when the application is installed. For more information about merging text files with the database, see Section 7.1.2, "Format of Registration Files."

7.1.1 Structure of the Database

The registration database is stored in binary format in a file named REG.DAT. This file is saved in the user's Windows directory.

Data in the registration database is in the form of a hierarchically structured tree. Each node in the tree is identified by a key name. Each key name is a string from the set of printable ASCII characters (values 32 through 127). Key names cannot include a space, a backslash (\), or a wildcard (* or ?). Key names beginning with a period (.) are reserved.

Any key name can also be associated with a text string that provides further information about that key. The text string can contain any character from the set of printable ASCII characters. These text strings are also called values.

Each key name is unique with respect to the key that is immediately above it in the hierarchy. For example, the **open** and **print** keys are often subkeys of the key named **shell**. Both **open** and **print** might have subkeys named **command**, but **open** could not have two subkeys named **command**.

The system defines a standard entry for the root level of the database: **HKEY_CLASSES_ROOT**. Root-level key names that begin with a period are reserved by the system. Database entries that are subordinate to the **HKEY_CLASSES_ROOT** key define types (or classes) of documents and the properties that are associated with these classes. Information stored under **HKEY_CLASSES_ROOT** is used by Windows shell applications and by OLE applications.

The following table shows the structure of a typical REG.DAT file. In this table, bold characters designate reserved words and italic characters designate words or phrases that vary with the registering application.

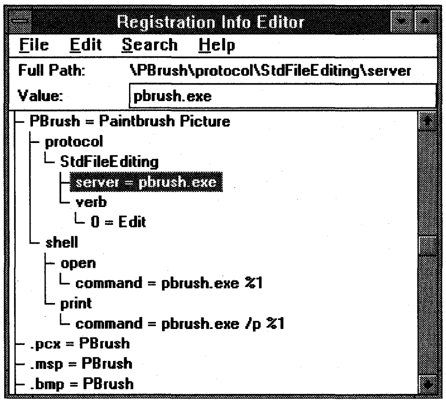
Key	Text string
HKEY_CLASSES_ROOT	
.ext	*class name*
ClassName	*class description*
shell	
open	
command	*command line to open application*
ddeexec	*DDE command used when opening document*
application	*DDE application name to start conversation*
topic	*topic of the DDE conversation*
ifexec	*DDE command if conversation does not start*
print	
command	*command line to open application*
ddeexec	*DDE command used when printing document*
application	*DDE application name to start conversation*
topic	*topic of the DDE conversation*
ifexec	*DDE command if conversation does not start*
protocol	
StdFileEditing	
server	*command line for opening application*
handler	*path and filename for handler DLL*
verb	*any verb*

Future versions of the database will include more reserved words. To avoid conflict with future versions, applications should record information that is not used by the Windows shell or OLE in private initialization files.

Standardized keys help an application navigate in the database. When an application has found the key for a feature, it typically uses the text string associated with that key. (As shown in the preceding list, however, not all keys have text strings.) For example, if an application needs to display the name of an application in a dialog box, the application might use the *ClassName* key to find the *class description* text string. The class name is often an abbreviated string, for application use only, whereas the class description is the full name of the application and is presented in the user interface.

Some standard entries to the database that are occasionally used by OLE server applications are not noted in the preceding list. For more information about these standard entries, see Chapter 6, "Object Linking and Embedding Libraries."

The following illustration shows how Windows Paintbrush is registered in REG.DAT (as displayed when REGEDIT.EXE is started with the **/v** option).



```
Full Path:   \PBrush\protocol\StdFileEditing\server
Value:       pbrush.exe

├ PBrush = Paintbrush Picture
│ ├ protocol
│ │ └ StdFileEditing
│ │   ├ server = pbrush.exe
│ │   └ verb
│ │     └ 0 = Edit
│ └ shell
│   ├ open
│   │ └ command = pbrush.exe %1
│   └ print
│     └ command = pbrush.exe /p %1
├ .pcx = PBrush
├ .msp = PBrush
├ .bmp = PBrush
```

7.1.2 Format of Registration Files

For most applications, the developer creates a registration (.REG) file that contains the database entries. Registration Editor (REGEDIT.EXE) can then be used to merge the .REG file into the user's REG.DAT file when the application is installed on the user's system.

The following example shows the format of a .REG file that would set up Microsoft Paintbrush with the entries shown in Figure 7.1:

```
REGEDIT

This is a comment line.

HKEY_CLASSES_ROOT\PBrush = Paintbrush Picture
HKEY_CLASSES_ROOT\.bmp = PBrush
HKEY_CLASSES_ROOT\.msp = PBrush
HKEY_CLASSES_ROOT\.pcx = PBrush
HKEY_CLASSES_ROOT\PBrush\shell\print\command = pbrush.exe /p %1
HKEY_CLASSES_ROOT\PBrush\shell\open\command = pbrush.exe %1
HKEY_CLASSES_ROOT\PBrush\protocol\StdFileEditing\verb\0 = Edit
HKEY_CLASSES_ROOT\PBrush\protocol\StdFileEditing\server = pbrush.exe
```

The first line of the file must be REGEDIT, as shown. Any subsequent lines that do not begin with **HKEY_CLASSES_ROOT** are currently treated as comments by REGEDIT.EXE. For compatibility with future versions of the database, however, a comment should not begin with a backslash (\) character or with the string HKEY. Each line to be added to the database must begin with a full key name. To create a key with an associated text string, the key name must be followed by at least one space, an equal sign (=), another space, and the string. Characters following the equal sign and single space are treated as the value of the key.

When SHELL.DLL encounters the string %1 in a command, it replaces that string with the name of the document being opened or printed.

A .REG file cannot be larger than 64K.

The setup procedure for the registering application typically merges this file with the user's REG.DAT file by running REGEDIT.EXE with the **/s** option. (Applications that must update the database with Windows 3.0 can use REGLOAD.EXE instead of REGEDIT.EXE to merge the files. REGLOAD.EXE is smaller than REGEDIT.EXE and does not require the common dialog box dynamic-link library COMMDLG.DLL.)

7.1.3 Class Registration

Database entries that are one level below the **HKEY_CLASSES_ROOT** root-level entry are defined as classes of documents. The exception to this definition is the *.ext* class.

Database entries that are subordinate to the class-definition entries describe the properties of a class. The database can describe two kinds of document properties for each class of document: shell properties and protocol properties.

7.1.3.1 Registering Filename Extensions

The *.ext* key name defines all files with that extension as members of a specified class. The registering application specifies the document class for an extension in the text string associated with the *.ext* key name.

Unlike other second-level key names, the *.ext* key name is not a class definition. Instead, it helps associate a class with a specific filename extension. For example, a word processor application can define a .DOC filename extension with the text string wpdoc. Then, when the word processor uses wpdoc as the class name for its documents, the .DOC extension is associated with that class.

The class name is the same name used by an OLE server application when it registers itself. For example, if a voice-annotation application named TALK.EXE registered as an OLE server, the information would look like this:

```
HKEY_CLASSES_ROOT\.tlk = Talk
HKEY_CLASSES_ROOT\Talk = Talk Voice Annotation
```

Filename extensions are recorded both in the database and in the [extensions] section of WIN.INI when the user records a filename association in the Associate dialog box. The Associate dialog box is displayed when the user chooses the Associate command from the File menu in File Manager. (Although File Manager automatically records the information in both places, SHELL.DLL does not. Applications that register filename extensions in the registration database should also record the information in WIN.INI, to provide compatibility with applications written before Windows 3.1.)

File Manager uses the filename associations recorded in WIN.INI if the information is not found in the registration database. If information is duplicated in the database and WIN.INI, File Manager uses the information in the database.

7.1.3.2 Shell Properties

Shell properties describe how a document of a given class interacts with Windows shell applications. There are two key names for shell properties: **open** and **print**. The **open** properties describe how the class responds to a request from a Windows shell application to open a document. The **print** properties describe how the class responds to a request from Print Manager to print a document.

Both the **open** and **print** key names must have the **command** subkey. The value assigned to **command** specifies the command line used to run the application. If appropriate, this value can include command-line options.

If an application supports DDE, it can also define the **ddeexec** subkey for either or both of the **open** and **print** key names. The text string given with the **ddeexec** key name is treated as a DDE command. Defining **ddeexec** is particularly useful if an application already supports DDE **open** and **print** commands. Using DDE messages can add flexibility, particularly for applications that support the multiple document interface (MDI), because a DDE message string can include more than one command.

The **ddeexec** key has three predefined subkeys: **application**, **topic**, and **ifexec**.

The text string given with the **application** key name specifies the application name to use in establishing the DDE conversation. If the registering application does not specify an **application** key, the shell uses the application name specified in the **command** key.

The text string given with the **topic** key name specifies the topic name of the DDE conversation. If the application does not register a **topic** key, the shell uses the System topic as the default topic name.

The text string given with the **ifexec** key name defines the DDE command to use when initiation of the DDE conversation fails (for example, if the application is not running). When the initiation fails, the command specified by the **command** key is carried out and then the string specified with the **ifexec** key is sent. (If an application does not specify a value for the **ifexec** key, the command specified by the **command** key is executed when initiation fails and the string specified with the **ddeexec** key is sent again.)

Opening Files An application should open a file in a new instance of the associated application, even if the application supports MDI. If the user has already opened the file, applications typically give the focus to the window with the file instead of obtaining a new copy of the file.

If an MDI application does not use memory efficiently when multiple instances of the application are running, the application can open the file in the existing instance, as a new MDI window.

Printing Files After opening the file as described in the preceding section, the application should carry out the **print** command. Whenever possible, applications should display the Print dialog box to give the user the opportunity to customize the print job. If this is not possible, the file should be printed immediately. Once the file is printed or the user chooses to cancel the print job, the application should close. (If the file was opened as a new MDI window, the application typically closes the window, rather than the entire application, when the print job has finished.)

7.1.3.3 Protocol Properties

A protocol is a convention for manipulating a document or some other collection of data. Database entries that are subordinate to the **protocol** key name describe the properties of a protocol. Although a class can support any number of protocols, currently only one is defined. This protocol, **StdFileEditing**, is used by documents that support OLE.

The **StdFileEditing** protocol has three subkeys: **server**, **handler**, and **verb**.

The text string given with the **server** key name is a command line that an OLE client application uses to start the server application for a linked or embedded object.

The text string given with the **handler** key name is the name of a dynamic-link library that acts as an object handler for OLE objects. For more information about object handlers, see Chapter 6, "Object Linking and Embedding Libraries."

The **verb** key name has subkeys that identify the kind of action a server should take when it opens an object. These subkeys are consecutive numbers, beginning with zero. The 0 subkey corresponds to the primary verb for the objects supported by the server. For example, 0 often means Edit and 1 often means Play. For more information about verbs, see Chapter 6, "Object Linking and Embedding Libraries."

For example, if an application named NewApp could not use REGEDIT.EXE to set up its **protocol** properties, it could set them up by using the following example:

```
HKEY hkProtocol;

if (RegCreateKey(HKEY_CLASSES_ROOT,             /* root              */
    "NewAppDocument\\protocol\\StdFileEditing", /* protocol string   */
    &hkProtocol) != ERROR_SUCCESS)             /* protocol key handle */
        return FALSE;

RegSetValue(hkProtocol,             /* handle to protocol key       */
    "server",                       /* name of subkey               */
    REG_SZ,                         /* required                     */
    "newapp.exe",                   /* command to activate server   */
    10);                            /* text string size             */

RegSetValue(hkProtocol,             /* handle to protocol key       */
    "handler",                      /* name of subkey               */
    REG_SZ,                         /* required                     */
    "nwappobj.dll",                 /* name of object handler       */
    12);                            /* text string size             */

RegSetValue(hkProtocol,             /* handle to protocol key       */
    "verb\\0",                      /* name of subkey               */
    REG_SZ,                         /* required                     */
    "Edit",                         /* server should edit object    */
    4);                             /* text string size             */

RegCloseKey(hkProtocol);            /* close protocol key and subkeys   */
```

7.1.3.4 Server Registration in WIN.INI

When an application creates a **server** protocol property and saves this key in REG.DAT, SHELL.DLL also puts this information into the WIN.INI initialization file. Some applications that use linked and embedded objects were developed before the implementation of the registration database. The information in WIN.INI allows such an application to find the command line that starts the server for an object. Server registration entries in WIN.INI are also written to the registration database whenever the user starts Windows.

The server registration entries in WIN.INI are in a section headed [embedding]. If an [embedding] section does not already exist when a registering application calls the **RegCloseKey** function for a key, SHELL.DLL creates it. When an application calls **RegCloseKey**, every class-definition key in REG.DAT that is not already in the [embedding] section is added to WIN.INI, not simply the key for which **Reg-CloseKey** was called.

The server information in WIN.INI is recorded in the following form:

[embedding]
*ClassName=comment,textual class name,path/arguments,*Picture

The keyword Picture indicates that the server can produce metafiles for use when rendering objects. Because commas are used as field separators, none of the fields can contain a comma.

A server can register only the name and arguments for its executable file, rather than the entire path, if the application is always installed in a directory that is mentioned in the PATH environment variable. Usually, registering the path and filename is less ambiguous than registering only the filename.

When the database is opened, the shell library reads the [embedding] section of WIN.INI and updates the registration database with any new information it contains. If the [embedding] section contains information that conflicts with REG.DAT, the information in REG.DAT is overwritten. When the database is closed, the shell library writes the information in REG.DAT back into the [embedding] section of WIN.INI. This ensures that applications that depend on WIN.INI for information about linked and embedded objects retrieve current information and that new OLE applications can simply read from and write to REG.DAT.

7.1.4 Querying and Deleting Database Entries

An application can use the **RegCreateKey** and **RegSetValue** functions to add keys to the registration database and the **RegCloseKey** function to indicate that a key is no longer needed by the application. Other registration functions allow an application to query the contents of the database and delete keys.

An application can use the **RegEnumKey** function to determine the subkeys of a specified key. Because the first parameter of **RegEnumKey** must be the handle of an open key, this function is typically preceded by a call to the **Reg-OpenKey** function and followed by a call to **RegCloseKey**. (Because the **HKEY_CLASSES_ROOT** key is always open, bracketing **RegEnumKey** with **RegOpenKey** and **RegCloseKey** is not strictly necessary when **HKEY_CLASSES_ROOT** is specified as the first parameter of **RegEnum-Key**. Using **RegOpenKey** and **RegCloseKey** is a time optimization in this case, however.) The **RegQueryValue** function retrieves the text string that has been associated with a key name.

The following example uses the **RegEnumKey** function to put the values associated with top-level keys into a list box:

```
HKEY hkRoot;
char szBuff[80], szValue[80];
static DWORD dwIndex;
LONG cb;

if (RegOpenKey(HKEY_CLASSES_ROOT, NULL, &hkRoot) == ERROR_SUCCESS) {
    for (dwIndex = 0; RegEnumKey(hkRoot, dwIndex, szBuff,
             sizeof(szBuff)) == ERROR_SUCCESS; ++dwIndex) {
        if (*szBuff == '.')
            continue;
        cb = sizeof(szValue);
        if (RegQueryValue(hkRoot, (LPSTR) szBuff, szValue,
                &cb) == ERROR_SUCCESS)
            SendDlgItemMessage(hDlg, ID_ENUMLIST, LB_ADDSTRING, 0,
                (LONG) (LPSTR) szValue);
    }
    RegCloseKey(hkRoot);
}
```

The following example uses the **RegQueryValue** function to retrieve the name of an object handler and then calls the **RegDeleteKey** function to delete the key if its value is nwappobj.dll:

```
char szBuff[80];
LONG cb;
HKEY hkStdFileEditing;

if (RegOpenKey(HKEY_CLASSES_ROOT,
        "NewAppDocument\\protocol\\StdFileEditing",
        &hkStdFileEditing) == ERROR_SUCCESS) {

    cb = sizeof(szBuff);
    if (RegQueryValue(hkStdFileEditing,
            "handler",
            szBuff,
            &cb) == ERROR_SUCCESS
            && lstrcmpi("nwappobj.dll", szBuff) == 0)
        RegDeleteKey(hkStdFileEditing, "handler");
    RegCloseKey(hkStdFileEditing);
}
```

7.2 Drag-Drop Feature

When an application implements the drag-drop feature, a user can select one or more files in File Manager, drag them to an open application, and drop them there.

The application in which the files were dropped receives a message it can use to retrieve the filenames and the coordinates of the point at which the files were dropped.

The drag-drop feature depends upon SHELL.DLL. The drag-drop feature does not depend in any way on the registration database, however.

An application that can accept dropped files from File Manager calls the **DragAcceptFiles** function for one or more of its windows. Then, when the user releases the mouse button to drop a file or files in the window specified in the call to **DragAcceptFiles**, File Manager sends the application a WM_DROPFILES message. (File Manager does not send the WM_DROPFILES message to an application unless the application calls **DragAcceptFiles**.) WM_DROPFILES contains a handle of an internal data structure the application can query to retrieve the name of the dropped file and the coordinates of the position at which the cursor was located when the file was dropped. The application can use the **DragQueryFile** function to retrieve the number of files that were dropped and their names. The **DragQueryPoint** function returns the window coordinates of the cursor when the user released the mouse button.

To free the memory allocated by the system for the WM_DROPFILES message, an application should call the **DragFinish** function when it is finished.

For example, an application can call the **DragAcceptFiles** function when it starts and call a drag-drop function when it receives a WM_DROPFILES message, as shown in the following example:

```
case WM_CREATE:
    DragAcceptFiles(hwnd, TRUE);
    break;

case WM_DROPFILES:
    DragFunc(hwnd, wParam);
    break;

case WM_DESTROY:
    DragAcceptFiles(hwnd, FALSE);
    break;
```

The following example uses the **DragQueryPoint** function to determine where to begin to write text. The first call to the **DragQueryFile** function determines the number of dropped files. The loop writes the name of each file, beginning at the point returned by **DragQueryPoint**.

```
POINT pt;
WORD cFiles, a;
char szFile[80];

DragQueryPoint((HANDLE) wParam, &pt);
```

```
cFiles = DragQueryFile((HANDLE) wParam, 0xFFFF, (LPSTR) NULL, 0);

for(a = 0; a < cFiles; pt.y += 20, a++) {
    DragQueryFile((HANDLE) wParam, a, szFile, sizeof(szFile));
    TextOut(hdc, pt.x, pt.y, szFile, strlen(szFile));
}

DragFinish((HANDLE) wParam);
```

7.3 Using Associations to Find and Start Applications

File Manager includes an Associate dialog box that makes it possible for users to associate a filename extension with a specific application. File Manager stores these associations in the registration database and the WIN.INI initialization file. If a file has a filename extension that has been associated with an application, that application starts automatically whenever a user double-clicks that file in File Manager.

Using the **FindExecutable** and **ShellExecute** functions, applications can take advantage of such associations to find and start applications or open and print files.

An application can use the **FindExecutable** function to retrieve the name and handle of the executable file that is associated with a specified filename. The **ShellExecute** function either opens or prints a specified file, depending on the value of its *lpszOp* parameter. To open a document file, the function relies on the association of the filename extension.

7.4 Extracting Icons from Executable Files

An application can use the **ExtractIcon** function to retrieve the handle of an icon from a specified executable file, dynamic-link library, or icon file. The following example uses the **DragQueryPoint** function to retrieve the coordinates of the point where a file was dropped, the **DragQueryFile** function to retrieve the filename of a dropped file, and the **ExtractIcon** function to retrieve the handle of the first icon in the file, if any:

```
HDC hdc;
HANDLE hCurrentInst, hicon;
POINT pt;
char szFile[80];

hCurrentInst = (HANDLE) GetWindowWord(hwnd, GWW_HINSTANCE);

DragQueryPoint((HANDLE) wParam, &pt);
```

```
DragQueryFile((HANDLE) wParam, 0, szFile, sizeof(szFile));
hicon = ExtractIcon(hCurrentInst, szFile, 0);

if (hicon == NULL)
    TextOut(hdc, pt.x, pt.y, "No icons found.", 15);
else if (hicon = (HICON) 1)
    TextOut(hdc, pt.x, pt.y,
        "File must be .EXE, .ICO, or .DLL.", 33);
else
    DrawIcon(hdc, pt.x, pt.y, hicon);
```

7.5 Related Topics

For more information about OLE, see Chapter 6, “Object Linking and Embedding Libraries.”

For more information about Program Manager, see Chapter 17, “Shell Dynamic Data Exchange Interface.”

Tool Helper Library

Chapter 8

The tool helper library (TOOLHELP.DLL) makes it easier for developers who work with the Microsoft Windows 3.1 operating system to obtain system information and control system activity. This dynamic-link library was designed to streamline the creation of Windows-hosted tools, specifically Windows-hosted debugging applications. TOOLHELP.DLL is available to applications running with Windows versions 3.0 and later.

To use the elements of TOOLHELP.DLL in an application, you must include the TOOLHELP.H header file in the application source files, link the application with TOOLHELP.LIB, and ensure that TOOLHELP.DLL is in the system path.

The following topics are related to the information in this chapter:

- Debugging
- Memory management
- Windows classes
- Task management
- Interrupts

8.1 Calling Tool Helper Functions

Most of the functions in TOOLHELP.DLL use structures to return information. The first member in each of these structures is a doubleword value named **dwSize**. This value must be initialized before an application calls the function that uses the structure; otherwise, the function fails.

The **dwSize** member enables new versions of TOOLHELP.DLL to include additional features without breaking code written for structures in Windows versions earlier than 3.1.

The THSAMPLE.C sample program demonstrates how to use some of the functions in TOOLHELP.DLL. For a full description of these functions, see the *Microsoft Windows Programmer's Reference, Volume 2*. For a full description of the TOOLHELP.DLL structures, see the *Microsoft Windows Programmer's Reference, Volume 3*.

8.2 Accessing Internal Windows Lists

TOOLHELP.DLL includes functions that enable you to retrieve information from the internal Windows lists. These lists include the class list, module list, and task queue.

8.2.1 Walking the Windows Class List

The **ClassFirst** function fills a **CLASSENTRY** structure with information about the first class on the Windows class list. This information includes the name of the class and the instance handle of the task that owns the class.

You use **ClassFirst** to begin a walk through the Windows class list. The **ClassNext** function continues the walk by filling a **CLASSENTRY** structure with information about the next class on the Windows class list.

You use the **GetClassInfo** function to obtain more specific class information. **GetClassInfo** requires the instance handle provided by **ClassFirst** or **ClassNext** in the **CLASSENTRY** structure.

8.2.2 Walking the Windows Module List

The **ModuleFirst** function fills a **MODULEENTRY** structure with information about the first module on the list of all currently loaded modules. This information includes the module name, handle, reference count, path to the executable file, and so on.

You use **ModuleFirst** to begin a walk through the Windows module list. The **ModuleNext** function continues the walk by filling a **MODULEENTRY** structure with information about the next module on the list.

The **ModuleFindHandle** function fills a **MODULEENTRY** structure with information about a module whose handle is known. The **ModuleFindName** function fills a **MODULEENTRY** structure with information about a module whose name is known. You use **ModuleFindHandle** or **ModuleFindName**, rather than **ModuleFirst**, to begin a walk through the Windows module list at a specific module, rather than at the first module on the list.

8.2.3 Walking the Windows Task Queue

The **TaskFirst** function fills a **TASKENTRY** structure with information about the first task in the Windows task queue. This information includes the task handle, SS register value, SP register value, stack dimensions, number of pending events, PSP offset, and so on.

You use **TaskFirst** to begin a walk through the Windows task queue. The **TaskNext** function continues the walk by filling a **TASKENTRY** structure with information about the next task in the task queue.

The **TaskFindHandle** function fills a **TASKENTRY** structure with information about a task whose handle is known. You use **TaskFindHandle**, rather than

TaskFirst, to begin a walk through the Windows task queue at a specific task, rather than at the first task in the queue.

8.3 Obtaining Advisory Information

To simplify system analysis, TOOLHELP.DLL includes functions that retrieve general information about the USER heap, GDI heap, memory manager, and virtual timer.

The **SystemHeapInfo** function fills a **SYSHEAPINFO** structure with information about the USER and GDI heaps. This information includes the percentage of free space and the segment handle for each heap.

The **MemManInfo** function fills a **MEMMANINFO** structure with status and performance information about the memory manager. This information includes the size of the largest free memory object, the maximum number of pages available, the maximum number of lockable pages, total linear space, total unlocked pages, number of pages in the system swap file, and so on.

The **TimerCount** function fills a **TIMERINFO** structure with the execution times of the current task and virtual machine (VM).

8.4 Walking the Global and Local Heaps

TOOLHELP.DLL includes functions that enable a developer to examine objects on the global and local heaps.

8.4.1 Walking the Global Heap

The **GlobalInfo** function fills a **GLOBALINFO** structure with information about the global heap. This information includes the total number of items, the number of free items, and the number of "least recently used" (LRU) items on the global heap. The information enables the application to determine how much memory to allocate for a global-heap walk. The application must allocate the memory before starting the walk. If the application allocates any memory after starting the walk, the results of the heap walk will be corrupt.

The **GlobalFirst** function fills a **GLOBALENTRY** structure with information about the first object on the global heap. This information includes the structure size, the size and address of the object, the lock count, and so on.

You use **GlobalFirst** to begin a walk through the global heap. The **GlobalNext** function continues the walk by filling a **GLOBALENTRY** structure with information about the next object on the global heap.

The **GlobalEntryHandle** function fills a **GLOBALENTRY** structure with information about a global object whose handle or selector is known. The **GlobalEntryModule** function fills a **GLOBALENTRY** structure with information about a specific segment in a module. You use **GlobalEntryHandle** or **GlobalEntryModule**, rather than **GlobalFirst**, to begin a walk through the global heap at a specific object, rather than at the first object on the global heap.

8.4.2 Walking the Local Heap

The **LocalInfo** function fills a **LOCALINFO** structure with the total number of items on the local heap. This information enables the application to determine how much memory to allocate for a local-heap walk. The application must allocate the memory before starting the walk. If the application allocates any memory after starting the walk, the results of the heap walk will be corrupt.

The **LocalFirst** function fills a **LOCALENTRY** structure with information about the first object on the local heap. This information includes the structure size; the handle, address, and size of the object; the lock count; and so on.

You can use **LocalFirst** to begin a walk through the local heap. The **LocalNext** function continues the walk by filling a **LOCALENTRY** structure with information about the next object on the local heap.

8.5 Tracing the Windows Stack

The **StackTraceFirst** function fills a **STACKTRACEENTRY** structure with information about the first stack frame for an inactive task. This information includes the stack-frame module handle, segment number, register contents, frame type, and so on.

You use **StackTraceFirst** to begin a stack trace of an inactive task. The **StackTraceNext** function continues the stack trace by filling a **STACKTRACEENTRY** structure with information about the task's next stack frame.

The **StackTraceCSIPFirst** function fills a **STACKTRACEENTRY** structure with information about a stack frame whose SS:BP and CS:IP values are known. You should use **StackTraceCSIPFirst**, rather than **StackTraceFirst**, to begin a stack trace of an active task.

8.6 Examining and Modifying Memory Contents

TOOLHELP.DLL includes functions that enable you to examine and modify global memory contents without consideration for selector tiling and aliasing or read-write attributes.

The **MemoryRead** function reads global memory at a specific selector and offset. The **MemoryWrite** function writes to global memory at a specific selector and offset.

The **GlobalHandleToSel** function converts a global memory handle to a selector.

8.7 Installing Callback Functions

TOOLHELP.DLL includes functions that enable you to trap an application's interrupts and notifications.

The **InterruptRegister** function installs a callback function that handles all system interrupts. The callback function must be reentrant and must explicitly preserve all register values. The **InterruptUnRegister** function restores the default processing.

The **NotifyRegister** function installs a notification callback function for a specific task. Typically, the notification callback function cannot use any Windows functions except the TOOLHELP.DLL functions and the **PostMessage** function. The **NotifyUnRegister** function restores the default processing.

The exit code returned by a non-Windows application may reflect an error encountered by Windows when it attempted to start the application, rather than a value returned by the application itself. These error values are as follows:

Error value	**Cause**
0x81	Could not start the application because of a file-access problem. This problem originated either in the application or its PIF file. Following are likely reasons for this error value: File not found Path not found No file handles Invalid drive Access denied Sharing violation Invalid executable format

Error value	Cause
0x82	Could not start the application, because of insufficient memory or disk space.
0x83	Abnormal termination.
0x84	Could not start the application, because of incorrect version.
0x85	Could not start the application, because MS-DOS Interrupt 21h Function 4B00h (Load and Execute Program) failed.
0x86	Could not start the application, because the TOOLHELP.DLL task-switching functions prevented it from starting.

8.8 Controlling Process Execution

TOOLHELP.DLL includes four functions you can use to control process execution: **TaskGetCSIP**, **TaskSetCSIP**, **TaskSwitch**, and **TerminateApp**. These functions are designed for use exclusively in Windows-hosted debuggers.

When an inactive task is activated, it begins execution at the location specified by its CS:IP value. The **TaskSetCSIP** function sets this value, and the **TaskGetCSIP** function returns the value.

The **TaskSwitch** function activates a specific task beginning at a specified CS:IP value.

The **TerminateApp** function terminates an application as if a general protection (GP) fault had occurred.

Data Decompression Library

Chapter 9

The Microsoft Windows operating system includes the dynamic-link library LZEXPAND.DLL. Typically, an application calls functions in LZEXPAND.DLL to decompress data previously compressed by Microsoft File Compression Utility (COMPRESS.EXE).

A version of LZEXPAND.DLL was shipped with Windows version 3.0. That version of LZEXPAND.DLL does not contain the full set of functions that is included with the Windows 3.1 version. Applications that could be installed on a system running Windows 3.0 should always check the version number of the library to ensure that the correct version is being used. For more information about checking version numbers, see Chapter 11, "File Installation Library."

This chapter describes important concepts relating to data compression and describes the decompression functions in LZEXPAND.DLL.

9.1 Data Compression

Data compression is an operation that reduces the size of a file by minimizing redundant data. In a file that contains text, redundant data could be frequently occurring characters, such as the space character, or common vowels, such as the letters *e* and *a*; it could also be frequently occurring character strings. Data compression operations create a compressed version of a file by minimizing this redundant data.

Each of the many types of data-compression operations minimizes redundant data in a unique manner. For example, the Huffman encoding algorithm assigns a code to characters in a file based on how frequently those characters occur. Another compression algorithm, called run-length encoding, generates a two-part value for repeated characters: The first part specifies the number of times the character is repeated, and the second part identifies the character. Another compression algorithm, known as the Lempel-Ziv algorithm, converts variable-length strings into fixed-length codes, which consume less space than the original strings.

To compress large applications or data files, you can run COMPRESS.EXE from the Microsoft MS-DOS® command line. COMPRESS.EXE uses the Lempel-Ziv compression algorithm.

9.2 Data Decompression

Applications can call the functions in LZEXPAND.DLL to decompress files compressed with COMPRESS.EXE. The functions can also process uncompressed files without attempting to decompress them.

The following table describes each function found in LZEXPAND.DLL:

Function	Purpose
CopyLZFile	Copies a source file to a destination file. If the source file was compressed, this function creates a decompressed destination file. If the source file was not compressed, this function duplicates the original file. This function is intended for multiple-file copy operations.
GetExpandedName	Retrieves the original name of a compressed file if the **/r** switch was used during compression of the file.
LZClose	Closes a file that was opened when the application called the **LZOpenFile** or the **OpenFile** function.
LZCopy	Copies a source file to a destination file. If the source file was compressed, this function creates a decompressed destination file. If the source file was not compressed, this function duplicates the original file. This function is intended for single-file copy operations.
LZDone	Frees memory allocated by the **LZStart** function. The **LZStart** and **LZDone** functions are used with the **CopyLZFile** function to copy multiple files.
LZInit	Creates structures that are used for decompressing files.
LZOpenFile	Opens a file. If the file was compressed, this function returns a special file handle that identifies the compressed file; if the file was not compressed, this function returns an MS-DOS file handle.
LZRead	Reads a specified number of bytes from a file. If the file was compressed, this function decompresses the bytes before copying them to the destination buffer.
LZSeek	Positions the file pointer within the decompressed image of a compressed file. The application calls this function to position the pointer prior to calling the **LZRead** function.
LZStart	This function allocates memory for multiple-file copy operations.

For more information about individual functions, see the *Microsoft Windows Programmer's Reference, Volume 2.*

9.3 Decompressing a Single File

An application can decompress a single compressed file by performing the following tasks:

1. Open the compressed file by calling the **LZOpenFile** function or a combination of the **OpenFile** and **LZInit** functions. For information about the **OpenFile** function, see the *Microsoft Windows Programmer's Reference, Volume 2.*
2. Open the destination file by calling the **LZOpenFile** or **OpenFile** function.
3. Copy the source file to the destination file by calling the **LZCopy** function and passing the handles returned by **LZOpenFile** (or **LZInit**).
4. Close the files by calling the **LZClose** function.

9.4 Decompressing Multiple Files

An application can decompress multiple files by performing the following tasks:

1. Open the source file by calling the **LZOpenFile** function or a combination of the **OpenFile** and **LZInit** functions.
2. Open the destination file by calling the **LZOpenFile** or **OpenFile** function.
3. Allocate memory for the copy operation by calling the **LZStart** function.
4. Copy the source files to the destination files by calling the **CopyLZFile** function.
5. Release the allocated memory by calling the **LZDone** function.
6. Close the files by calling the **LZClose** function.

9.5 Reading Bytes from Compressed Files

In addition to decompressing a complete file at a time, an application can decompress compressed files a portion at a time by using the **LZSeek** and **LZRead** functions. These functions are particularly useful when it is necessary to extract parts of large files. For example, a font manufacturer may have compressed files containing font metrics in addition to character data. To use the information in these files, an application would need to decompress the file; however, most applications would use only part of the file at any particular time. When the user queried the font metrics, the application would extract data from the header. When the user rendered text output, the application would reposition the file pointer by calling **LZSeek** and extract the character data.

System Resources Stress-Testing Library

Chapter 10

The system resources stress-testing library (STRESS.DLL) is a dynamic-link library that artificially consumes system resources, enabling developers to observe how an application behaves in scarce-resource conditions. This library was designed to make scarce-resource testing easier and more realistic. It is used by the STRESS.EXE utility.

10.1 System Resources Stress-Testing Library Functions

Following are the system resources affected by STRESS.DLL, with the functions that consume and release each resource:

Resource	Allocation function	Release function
Global memory	**AllocMem**	**FreeAllMem**
GDI heap memory	**AllocGDIMem**	**FreeAllGDIMem**
User heap memory	**AllocUserMem**	**FreeAllUserMem**
Disk space	**AllocDiskSpace**	**UnAllocDiskSpace**
File handles	**AllocFileHandles**	**UnAllocFileHandles**

For more information about STRESS.DLL functions, see the *Microsoft Windows Programmer's Reference, Volume 2.*

File Installation Library

Chapter 11

The file installation library in the Microsoft Windows version 3.1 operating system makes it easier for applications to install files properly and enables utility programs to analyze files that are currently installed.

The following topics are related to the information in this chapter:

- Resources
- Microsoft Windows Resource Compiler (RC)

11.1 File Installation Concepts

The file installation library includes functions that determine where a file should be installed, identify conflicts with currently installed files, and perform the installation process. These functions enable installation programs to avoid the following problems:

- Installing older versions of components over newer versions
- Changing the language in a mixed-language system without notification
- Installing multiple copies of a library in different directories
- Copying files to network directories shared by multiple users

The file installation library also includes functions that enable applications to query a version resource for information about a file and present the information to the user in a clear format. This information includes the file's purpose, author, version number, and so on. (For more information about version resources, see Section 11.3, "Adding Version Information to a File.")

The file installation library is available for Windows and non-Windows applications. Windows applications should use the dynamic-link library VER.DLL and the header file VER.H. Non-Windows applications should use one of the following static-link libraries: VERS.LIB, VERC.LIB, VERM.LIB, or VERL.LIB. Applications that use the static-link libraries should use the following line before including VER.H:

```
#define LIB
```

11.2 Creating an Installation Program

An installation program typically has the following goals:

- To place files in the correct location
- To notify the user if the installation program is replacing an existing file with a version that is significantly different—for example, replacing a German file with an English file, or replacing a newer file with an older file

When writing the installation program, you must have the following information for each file on the installation disk(s):

- The name and location of the file (referred to as the source file).
- The name of the equivalent file on the user's hard disk (referred to as the destination file). This name is usually the same as the filename on the installation disk.
- The sharing status of the file—that is, whether the file is private to the application being installed or could be shared by multiple applications.

For each file on the installation disk(s), the installation program must, at least, call the **VerFindFile** and **VerInstallFile** functions. These functions are described briefly in the rest of this section.

You use the **VerFindFile** function with the destination-file name to determine where the file should be copied to on the disk. This function also enables you to specify whether the file is private to the application or can be shared. If a problem occurs in finding the file, **VerFindFile** returns an error value. For example, if Windows is using the destination file, **VerFindFile** returns VFF_FILEINUSE. The installation program must notify the user of the problem and respond to the user's decision to continue or end the installation.

The **VerInstallFile** function copies the source file to a temporary file in the directory specified by **VerFindFile**. If necessary, **VerInstallFile** expands the file by using the functions in the data decompression library, LZEXPAND.DLL.

VerInstallFile compares the version information of the temporary file to that of the destination file. If they differ, **VerInstallFile** returns one or more error values. For example, it returns VIF_SRCOLD if the temporary file is older than the destination file and VIF_DIFFLANG if the files have different language identifiers or code-page values. The installation program must notify the user of the problem and respond to the user's decision to continue or end the installation.

Some **VerInstallFile** errors are recoverable. That is, the installation program can call **VerInstallFile** again, specifying the VIFF_FORCEINSTALL option, to install the file regardless of the version conflict. If **VerInstallFile** returns VIF_TEMPFILE and the user chooses not to force the installation, the installation program should delete the temporary file.

VerInstallFile could encounter a nonrecoverable error when attempting to force installation, even though the error did not exist previously. For example, the file could be locked by another user before the installation program tried to force installation. If an installation program attempts to force installation after a nonrecoverable error, **VerInstallFile** fails. The installation program must deal with this situation.

The recommended solution is to display a common dialog box with the buttons Install, Skip, and Install All for all errors. The Install All button should prevent the installation program from prompting the user about similar errors by including the VIFF_FORCEINSTALL option in all subsequent uses of **VerInstallFile**. For nonrecoverable errors, the Install and Install All buttons should be disabled.

To display a useful error message to the user, the installation program usually must retrieve information from the version resources of the conflicting files. The file installation library provides four functions the installation program can use for this purpose: **GetFileVersionInfoSize**, **GetFileVersionInfo**, **VerQueryValue**, and **VerLanguageName**. The **GetFileVersionInfoSize** function returns the size of the version information. The **GetFileVersionInfo** function then uses information retrieved by **GetFileVersionInfoSize** to retrieve a structure that contains the information. The **VerQueryValue** function retrieves a specific member from that structure.

For example, if **VerInstallFile** returns the VIF_DIFFTYPE error, the installation program should use **GetFileVersionInfoSize**, **GetFileVersionInfo**, and **VerQueryValue** on the temporary and destination files to obtain the general type of each file. If the languages of the files conflict, the installation program should also use the **VerLanguageName** function to translate the binary language identifier into a text representation of the language. (For example, 0x040C translates to the string French.)

If **VerInstallFile** returns a file error, such as VIF_ACCESSVIOLATION, the installation program should use MS-DOS Interrupt 21h Function 59h (Get Extended Error) to obtain the most recent error value. The program should translate this value into an informative message to display to the user. The program must not yield control between calling **VerInstallFile** and calling Get Extended Error. If it does, the MS-DOS error value could reflect a later error. (An error could also occur while the program is making the MS-DOS call.)

For more information about the version-stamping functions, see the *Microsoft Windows Programmer's Reference, Volume 2*. For more information about Interrupt 21h Function 59h, see the *Microsoft MS-DOS Programmer's Reference.*

11.3 Adding Version Information to a File

Version information can be added to any Windows file that can have Windows resources, such as a dynamic-link library, an executable file, or a font file. To add the information, you must create a version resource and add the resource to the file by using RC. For more information about using RC, see *Microsoft Windows Programming Tools.*

32-Bit Memory Management Library

Chapter 12

One of the significant features of 80386 and 80486 processors 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 the Microsoft Windows operating system continues to adhere to a segmented 16-bit memory model, Windows does provide a set of functions that allow an application to make use of the 32-bit memory-addressing capabilities of the 80386 and 80486 processors. These functions are available to an application through a dynamic-link library (DLL) named WINMEM32.DLL.

Your application's installation program should use the file installation library (VER.DLL) to ensure that it does not install an older version of WINMEM32.DLL over a newer version. For more information about VER.DLL, see Chapter 11, "File Installation Library."

This chapter 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:

- Some of the differences between a segmented memory model and a flat memory model
- Use of WINMEM32.DLL to take advantage of the 32-bit memory-addressing capabilities of 80386 and 80486 processors
- Programming considerations for use of 32-bit memory in a Windows application
- Use of 32-bit memory in a Windows application
- A directory of WINMEM32.DLL functions
- Assembly-language examples illustrating how to use WINMEM32.DLL functions

Important You should be thoroughly familiar with the following information about 80386 and 80486 processors that is not covered in this chapter:

- Terminology and concepts relating to the architecture
- Code-management features
- Memory-management features

Only developers with experience writing Windows applications and assembly language code should attempt to use these functions in an application.

12.1 Segmented and Flat Memory Models

The family of processors that includes 80286, 80386, and 80486 processors implements a segmented memory model in which system memory is divided into 64K segments. In the real mode of these processors, the address of any byte consists of two 16-bit values: a segment address and an offset. (Windows version 3.1 does not support real mode.) In the protected mode of the 80286, 80386, and 80486 processors, the segment address is replaced by a selector value that the processor uses to access the 64K segment. In either mode, a memory object larger than 64K occupies all or part of several segments. An application cannot access such an object as though it consisted 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 processors. 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, data, or both occupy 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 offset portion of the memory throughout the address space without having to deal with multiple segment boundaries.

To a certain extent, the flat memory model most closely resembles the tiny memory model, in which both code and data occupy a single segment; of course, 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.

The flat memory model makes it possible for you to ignore segments and segment registers. The segment registers are loaded at the start of the 32-bit code and are then left alone. The rest of the application runs in this purely 32-bit offset mode—all pointers are near pointers.

It is not possible to implement a Windows application by using an exclusively flat memory model. Because Windows itself relies on the 16-bit segmented memory model, any application that interacts with Windows must implement at least one 16-bit code segment. Despite this limitation, 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 cooperates fully with Windows and similar platforms. For more information, see Section 12.3.1, "Flat Memory Model Limitations."

12.2 Using the WINMEM32.DLL Library

Although you could directly implement code for a flat memory model 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 MS-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 standard or 386 enhanced mode. However, because the 32-bit registers of the 80386 or 80486 processor are available only when Windows is in 386 enhanced mode, WINMEM32.DLL is enabled only in that mode. If your application runs in standard mode, you must design your application so that it can access 16-bit memory instead of 32-bit memory. You can find out which mode Windows is running in by calling the **GetWinFlags** function.

WINMEM32.DLL contains eight functions that enable your application to access 32-bit memory. The following table summarizes each of these functions:

Function	Description
GetWinMem32Version	Returns the version number of the WINMEM32.DLL application programming interface (API).
Global16PointerAlloc	Converts a 32-bit pointer to a 16-bit pointer.
Global16PointerFree	Frees a pointer alias created by the **Global16PointerAlloc** function.
Global32Alloc	Allocates a 32-bit memory object.
Global32CodeAlias	Creates a code-segment alias for a 32-bit memory object, allowing code in the object to be executed.
Global32CodeAliasFree	Frees a code-segment alias created by the **Global32CodeAlias** function.
Global32Free	Frees a 32-bit memory object.
Global32Realloc	Changes the size of a 32-bit memory object.

A directory listing of these functions appears later in this chapter.

Because WINMEM32.DLL is a standard Windows DLL, your application loads it as it would any other DLL. Your application should be linked so that the case of the DLL entry point names is ignored.

The WINMEM32.DLL functions use the same calling conventions as other Windows functions. The DLL entry points are external **FAR PASCAL** procedures. They preserve the SS, BP, DS, SI, and DI registers, and they return values in the AX register or the DX:AX register pair.

12.3 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 selector, combined with a 16-bit offset within the segment. An application using the 32-bit registers of the 80386 or 80486 processor cannot directly call the Windows functions, because its far pointers are in the form 16:32 and Windows cannot work 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 makes calls to MS-DOS, to other DLLs, or to any other code that uses 16:16 pointers exclusively.

An important limitation on this helper segment is that it must not be discardable (although it can be movable). If the segment is discarded and a 32-bit segment attempts to access the segment, an indirect call into the Windows kernel module to reload the segment results. Because the source of this indirect call is not a 16-bit segment, the system might crash.

Another important consideration is that in writing your application you must not assume anything about the state of the 32-bit registers around 16:16 function calls. For instance, the Windows function calls preserve SI and DI registers, but they presently do not preserve ESI and EDI registers. If the application needs to preserve 32-bit registers around 16:16 function calls, it must explicitly push and pop the register values around the calls. If the 32-bit code segment that calls a Windows function (by means of the helper segment) needs ESI and EDI registers to 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 requires 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:

```
movzx    eax,ax       ; unsigned return

movsx    eax,ax       ; signed return
```

All these considerations apply equally to calls to Windows DLLs, MS-DOS, and other 16-bit functions.

12.3.1 Flat Memory Model Limitations

In the Windows environment, system memory is a shared resource that 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 systemwide memory locations. In other words, the selector that the application receives does not refer to linear address 0. This means that offset 400h for the selector does not point to the MS-DOS ROM BIOS data area, for example.

Windows applications do not need to address these systemwide memory locations directly, so there is no need to map these locations in the 32-bit memory objects.

12.3.2 The Application Stack

Windows cannot operate in an environment of mixed segment types (including both 16:16 and 16:32 segments). As a result, the stack selector size must match the corresponding code selector size. When the processor is executing code in a 16:32 (USE32) code segment, the selector in the SS register must contain a 16:32 selector. When the processor is executing code in a 16:16 (USE16) segment, the SS register must contain a 16:16 selector.

When the 80386 or 80486 processor is executing on a USE16 stack segment, it uses the low-order 16 bits of the ESP register as the SP register. Because only the low-order 16 bits are of use when the processor is running on a USE16 stack segment, the processor does not control how the high-order 16 bits of the ESP register are set. As a result, the high-order 16 bits are set at random. When an application switches to a USE32 stack segment, the ESP register contains a corrupted pointer unless the high-order 16 bits of ESP are set properly.

Suppose that 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, the application continues to use its USE32 stack. The USE16 code segment calls a Windows function, which changes the selector in the SS register to a USE16 selector. Because the stack is now USE16, the high-order 16 bits of the ESP register are set at random. The code that originally switched stacks then restores the original selector in SS and, lacking the information that the selector 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. One solution is for an application to 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 that used in C. Another solution is to maintain one stack but two stack selectors, one USE16 and the other USE32, both of which point to the same memory. This requires 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 switch back to the USE16 stack before doing anything else.

Because 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 to the USE32 stack selector. It sets these bits by placing the selector into the SS register, 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
```

12.3.3 Interrupt-Time Code

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. The code must use a USE16 stack. Data used at interrupt time must be USE16 data. This rule also applies to processor exceptions (such as the coprocessor exception) because they are handled as interrupts are handled. Note, however, that it is acceptable for a 32-bit code segment to access data in a USE16 data segment.

12.3.4 Programming Languages

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 write the helper segment in C, you must add assembly-language support for the more difficult tasks. In most cases, it is easier and more efficient to write the entire helper segment in assembly language.

12.4 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 remaining topics in this section briefly describe these uses.

12.4.1 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 does not require a dedicated helper segment because it contains no USE32 code segments. Instead, each 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 calls the **Global16Pointer-Alloc** function so that the application's USE16 code segment can perform the aliasing of 32-bit pointers to the 16-bit pointers.

12.4.2 Using 32-Bit Code and Data in a Subroutine Library

Using 32-bit segments for code and data can simplify porting an application from a 32-bit platform 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 MS-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 Section 12.3, "Considerations for Using 32-Bit Memory," 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.

12.4.3 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 to helper status exclusively. 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 then on, the USE32 segment takes control of the program, calling into the USE16 helper segment only when the application needs to call Windows or MS-DOS functions. The USE32 segment continues to control the flow of the program until the application is ready to close. Only then does it return control to the USE16 segment so the USE16 segment can free the 32-bit memory and perform other cleanup tasks before the application quits.

12.5 Error Values

This section describes error values returned by the functions that applications can use for 32-bit memory management. Most of these functions return zero to indicate success. The following table describes each error value:

Value	Meaning
WM32_Insufficient_Mem	Insufficient memory. There is not enough memory to satisfy the requested allocation or reallocation.
WM32_Insufficient_Sels	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.
WM32_Invalid_Arg	Invalid parameter. One of the parameters was invalid. For example, a size parameter might be out of range.
WM32_Invalid_Flags	Invalid flag. The *wFlags* parameter contained at least one invalid bit setting. The *wFlags* parameter currently is not used and must be set to zero.
WM32_Invalid_Func	Invalid function. The current Windows mode does not support this function. Windows supports the 32-bit memory functions only in 386 enhanced mode.

Floating-Point–Emulation Library

Chapter 13

This chapter describes two methods that can be used to support floating-point emulation in Windows applications. In particular, the chapter describes in detail the Windows 80x87 floating-point emulator in the dynamic-link library WIN87EM.DLL. This information is intended to be used by compiler vendors who want to develop floating-point emulators that are compatible with WIN87EM.DLL.

13.1 Emulation Methods

With floating-point emulation, Windows applications that contain floating-point instructions can run on any computer, regardless of whether the computer has floating-point hardware.

To support floating-point emulation for Windows applications, compiler vendors can use one of the following methods:

- Emulation by exception handler
- Windows 80x87 floating-point emulation

13.1.1 Emulation by Exception Handler

With emulation by exception handler, a Windows application contains floating-point instructions for all floating-point operations and an exception handler for occurrences of Interrupt 07h (coprocessor not available). When the application starts, it installs the exception handler and the exception handler processes any floating-point exceptions that occur thereafter.

When the application runs on a computer with no floating-point hardware, a floating-point exception occurs the first time a floating-point instruction is executed. The exception handler is responsible for patching and then restarting the instruction. To patch the floating-point instruction, the exception handler actually replaces it with a call to emulation code. The new instruction calls the emulation code directly (rather than generating an exception) for as long as the patched instruction remains in memory.

This method can be used only with the Microsoft Windows operating system, version 3.1, because Windows version 3.0 standard mode does not save and restore the state of the exception handler across task switches.

This method may be less efficient than other methods because it requires that floating-point instructions be patched while the application is running rather than while it is loading. As long as the patched instructions remain in memory, however, this method is as efficient as other methods. If Windows discards the code segments that contain the patched instructions, the floating-point instructions must be patched again because Windows always loads a fresh copy of the code when it restores the discarded segments.

13.1.2 Windows 80x87 Floating-Point Emulation

With Windows 80x87 floating-point emulation, the Windows application contains calls to floating-point instructions for all floating-point operations, but the application also includes fixup records for each instruction. When Windows loads the application, Windows determines whether floating-point hardware is present. If the hardware it is not present, Windows uses the fixup records to replace the actual instructions with calls to emulation code.

To support this method, the application's startup routine must check whether WIN87EM.DLL is present. Then the routine must initialize WIN87EM.DLL by calling the **__fpmath** function with the BX register set to 0 and must set the floating-point exception handler by calling the **__fpmath** function with the BX register set to 3 and the DS:AX registers pointing to the exception handler. When the application's **WinMain** function returns to the startup routine, the routine must release WIN87EM.DLL by calling the **__fpmath** function with the BX register set to 2. After WIN87EM.DLL has been released, the startup routine can end the application.

For this method to work correctly, the Windows application must contain the proper fixup records—sometimes called operating system (OS) fixups—to convert instructions to emulation calls. For WIN87EM.DLL, each call consists of an interrupt (**int**) instruction followed by one or more words defining the floating-point operation and operands. The call is actually generated by the addition of fixup values to the first two words of the corresponding floating-point instruction. The fixup values to use depend on the instruction—the values are defined as follows:

```
fINT    equu    0CDh
fFWAIT  equ     09Bh
fESCAPE equ     0D8h
fFNOP   equ     090h
fES     equ     026h
fCS     equ     02Eh
fSS     equ     036h
fDS     equ     03Eh
BEGINT  equ     034h
```

```
FIARQQ  equu    (fINT + 256*(BEGINT + 8)) - (fFWAIT + 256*fDS)
FISRQQ  equ     (fINT + 256*(BEGINT + 8)) - (fFWAIT + 256*fSS)
FICRQQ  equ     (fINT + 256*(BEGINT + 8)) - (fFWAIT + 256*fCS)
FIERQQ  equ     (fINT + 256*(BEGINT + 8)) - (fFWAIT + 256*fES)
FIDRQQ  equ     (fINT + 256*(BEGINT + 0)) - (fFWAIT + 256*fESCAPE)
FIWRQQ  equ     (fINT + 256*(BEGINT + 9)) - (fFNOP  + 256*fFWAIT)
FJARQQ  equ     256*(((0 shl 6) or (fESCAPE and 03Fh)) - fESCAPE)
FJSRQQ  equ     256*(((1 shl 6) or (fESCAPE and 03Fh)) - fESCAPE)
FJCRQQ  equ     256*(((2 shl 6) or (fESCAPE and 03Fh)) - fESCAPE)
```

Each of the six fixup record types consists of two one-word values, as shown in the following example:

```
osfixuptbl  label word
    DW  FIARQQ, FJARQQ
    DW  FISRQQ, FJSRQQ
    DW  FICRQQ, FJCRQQ
    DW  FIERQQ, 0h
    DW  FIDRQQ, 0h
    DW  FIWRQQ, 0h
osfixuptbllen = $-osfixuptbl
```

The loader assumes that each floating-point instruction is preceded by a **wait** instruction. The loader adds the first word to the combination of the **wait** instruction byte and the first byte in the floating-point instruction. For fixup types 1 through 3, the loader adds the second word to the second and third bytes of the floating-point instruction. For types 4 through 6, the loader makes no changes to these bytes (it adds zero).

Because WIN87EM.DLL polls for exceptions by using the **fwait** instruction, the loader must replace each **nop** and **fwait** instruction pair with a call to emulation code, even if a floating-point coprocessor is available. These instructions must have a corresponding fixup record of type 6.

WIN87EM.DLL does not emulate the following floating-point instructions:

fbld	**fsave**
fbstp	**fsetpm**
fcos	**fsin**
fdecstp	**fsincos**
fincstp	**fstenv**
finit	**fucom**
fldenv	**fucomp**
fnop	**fucompp**
fprem1	**fxtract**
frstor	

13.2 Windows 3.0 Limitations

Windows 3.0 does not correctly save and restore the emulator state for emulator functions 0x38 through 0x3E. This means that Windows applications that use a floating-point emulator other than WIN87EM.DLL may not run successfully if another application that is using WIN87EM.DLL is also running.

Windows 3.1 does correctly save and restore the emulator state. Therefore, applications that use other floating-point emulators should be run only under Windows 3.1.

13.3 Functions

This section describes the functions that can be used for 80x87 floating-point emulation.

_FPInit

2.x

LPVOID _FPInit(void)

The **_FPInit** function initializes the Windows floating-point–emulation library (WIN87EM.DLL) or floating-point coprocessor and sets up a default floating-point exception-handler routine. Only dynamic-link libraries (DLLs) need to call this function.

Parameters

This function has no parameters.

Return Value

The return value is a pointer to the previous floating-point exception handler.

Comments

A DLL must ensure that the floating-point emulator or coprocessor has been initialized before making any function calls that use floating-point arithmetic. If a task that does not initialize the floating-point emulator or coprocessor can call the DLL, or if the task's floating-point exception handler does not handle floating-point exceptions appropriately for the DLL, the DLL must call the **_FPInit** function to initialize the emulator or coprocessor. Before returning control to the calling task, the DLL must call the **_FPTerm** function to restore the previous exception handler.

See Also

_FPTerm

__fpmath

2.x

```
extern      __fpmath:far

mov         bx, Function        ; floating-point function
call        __fpmath            ; floating-point math
```

The **__fpmath** function is the control function for Windows 80x87 floating-point emulation.

Parameters

Function

Specifies the floating-point function to execute. The *Function* parameter can be one of the following values:

Value	Meaning
0	Initializes the floating-point emulator. An application calls this function when it starts. If an error occurs, the function sets the carry flag. Otherwise, it clears the flag.
1	Resets the floating-point emulator. The action carried out by this function is similar to the action carried out by the **finit** instruction.
2	Stops the floating-point emulator. An application called this function just before it ended.
3	Sets the handler for the coprocessor error exception (Interrupt 16). The DS:AX registers must contain the 32-bit address of the exception handler. The emulator calls the handler whenever an unmasked floating-point exception occurs. The exception handler can carry out any action—it does not have to return.
10	Retrieves a count of the elements on the floating-point stack, copying the count to the AX register. The number of elements is equal to the number of floating-point values on the floating-point coprocessor (if one is present) plus any additional values stored by the emulator.
11	Indicates whether a floating-point coprocessor is present. This function returns 1 in the AX register if a coprocessor is present. Otherwise, it returns 0.

Comments

Function values 4 through 9 are not used.

Example

The following example initializes the floating-point emulator:

```
xor      bx, bx           ; bx = 0 to initialize floating point
call     __fpmath
```

_FPTerm

2.x

void _FPTerm(*lpOldFPSigHandler*)
FARPROC *lpOldFPSigHandler*;   /* address of exception handler   */

The **_FPTerm** function restores the floating-point exception-handler routine that was in effect when a dynamic-link library (DLL) called the **_FPInit** function to initialize the floating-point emulator or coprocessor. Only DLLs need to call this function.

Parameters

lpOldFPSigHandler
Specifies the address of the previous exception handler.

Return Value

This function does not return a value.

Comments

A DLL must ensure that the floating-point emulator or coprocessor has been initialized before making any function calls that use floating-point arithmetic. If a task that does not initialize the floating-point emulator or coprocessor can call the DLL, or if it is possible that the task's floating-point exception handler does not handle floating-point exceptions appropriately for the DLL, the DLL must call the **_FPInit** function to initialize the emulator or coprocessor. Before returning control to the calling task, the DLL must call the **_FPTerm** function to restore the previous exception handler.

See Also

_FPInit

__Win87EmInfo

3.1

int __Win87EmInfo(*pWIS*, *cbWin87EmInfoStruct*)
Win87EmInfoStruct far **pWIS*;   /* buffer to receive information   */
int *cbWin87EmInfoStruct*;   /* size of buffer, in bytes   */

The **__Win87EmInfo** function retrieves information about the floating-point emulator, such as whether a floating-point coprocessor is present and the code and data segment addresses of the emulator.

Parameters

pWIS
Points to the **Win87EmInfoStruct** structure that is to receive the floating-point emulator information. The **Win87EmInfoStruct** structure has the following form:

```
typedef  struct _Win87EmInfoStruct {
    unsigned Version;
    unsigned SizeSaveArea;
    unsigned WinDataSeg;
    unsigned WinCodeSeg;
    unsigned Have80x87;
    unsigned Unused;
} Win87EmInfoStruct;
```

For more information about this structure, see Section 13.4, "Structures."

cbWin87EmInfoStruct
Specifies the size, in bytes, of the structure that is to receive the information.

Return Value This function returns zero if no errors occur. Otherwise, it returns a nonzero value.

__Win87EmRestore

3.1

int __Win87EmRestore(void far **pWin87EmSaveArea*, **int** *cbWin87EmSaveArea*)
void far **pWin87EmSaveArea*; /* buffer containing state */
int *cbWin87EmSaveArea*; /* size, in bytes, of buffer */

The **__Win87EmRestore** function restores the states of the floating-point coprocessor (if one is present) and the floating-point emulator to the states previously saved by the **__Win87EmSave** function.

Parameters

pWin87EmSaveArea
Points to the **Win87EmSaveArea** structure containing the state of the floating-point coprocessor and emulator. The **__Win87EmSave** function must have been used previously to fill the structure.

cbWin87EmSaveArea
Specifies the size, in bytes, of the structure containing the emulator state.

Return Value This function returns zero if the function is successful. Otherwise, it returns a nonzero value.

See Also **__Win87EmSave**

__Win87EmSave

3.1

int __Win87EmSave(*pWin87EmSaveArea***,** *cbWin87EmSaveArea***)**
void far ****pWin87EmSaveArea*;** /* buffer to receive state */
int *cbWin87EmSaveArea***;** /* size, in bytes, of buffer */

The **__Win87EmSave** function saves the current states of the floating-point coprocessor (if one is present) and the floating-point emulator, copying the states to the buffer pointed to by *pWin87EmSaveArea.*

An application that calls **__Win87EmSave** should call the **__Win87EmRestore** function before carrying out any floating-point operations.

Parameters

pWin87EmSaveArea
Points to the **Win87EmSaveArea** structure that is to receive the state of the floating-point emulator.

cbWin87EmSaveArea
Specifies the size, in bytes, of the structure to receive the emulator state.

Return Value

This function returns zero if the function is successful. Otherwise, it returns a nonzero value.

Comments

An application can find out the size, in bytes, of the buffer needed to save the floating-point states by using the **__Win87EmInfo** function to retrieve the **Win87EmInfoStruct** structure. The **SizeSaveArea** member of this structure specifies the size of the buffer.

See Also

__Win87EmInfo, **__Win87EmRestore**

13.4 Structures

This section describes the structures that can be used for 80x87 floating-point emulation.

Win87EmInfoStruct

3.1

```
typedef  struct _Win87EmInfoStruct {
    unsigned Version;
    unsigned SizeSaveArea;
    unsigned WinDataSeg;
    unsigned WinCodeSeg;
    unsigned Have80x87;
    unsigned Unused;
} Win87EmInfoStruct;
```

The **Win87EmInfoStruct** structure contains information about the floating-point emulator.

Members

Version
Specifies the major and minor version numbers. The high-order byte specifies the major version number, the low-order byte the minor version number.

SizeSaveArea
Specifies the size, in bytes, of the buffer needed to save the floating-point emulator state. An application uses the specified size to allocate sufficient space to save the state before calling the **__Win87EmSave** function.

WinDataSeg
Specifies the emulator's data segment address or selector.

WinCodeSeg
Specifies the emulator's code segment address or selector.

Have80x87
Specifies the floating-point emulator flag. If an 80287 or 80387 floating-point coprocessor is present, this member is 1. Otherwise, it is 0.

Unused
Not used.

See Also **__Win87EmInfo**, **__Win87EmSave**

Win87EmSaveArea

3.1

```
typedef  struct _Win87EmSaveArea {
    unsigned char  Save80x87Area[SIZE_80X87_AREA];
    unsigned char  SaveEmArea[];
} Win87EmSaveArea;
```

The **Win87EmSaveArea** structure contains the states of the floating-point coprocessor and floating-point emulator.

Members

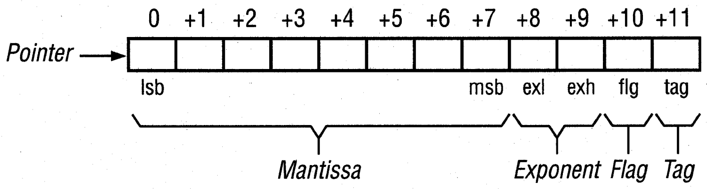
Save80x87Area
Specifies an array of values defining the state of the floating-point coprocessor if one is present. The array has the same format as data saved by an **fsave** instruction and consists of SIZE_80X87_AREA (94) array elements.

SaveEmArea
Specifies an array of values defining the state of the floating-point emulator. The array has the following form:

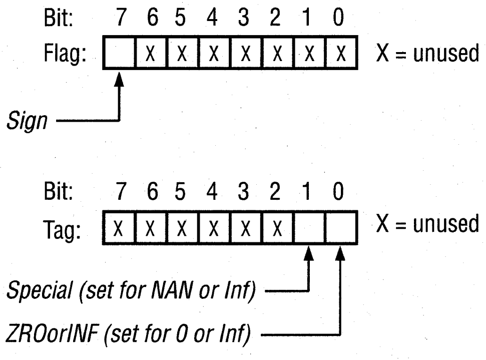
```
Have8087     db  0        ; 1 if coprocessor is present; otherwise, 0
             db  ?        ; reserved
             dw  ?        ; reserved
             dw  ?        ; reserved

ControlWord  label   word ; emulator control word
CWmask       db      ?    ; exception masks
CWcntl       db      ?    ; arithmetic control flags

StatusWord   label   word ; emulator status word
SWerr        db      ?    ; exception flags
SWcc         db      ?    ; condition codes

BASstk       dw      ?    ; offset of start of emulator register area
CURstk       dw      ?    ; offset of current top-of-stack register
LIMstk       dw      ?    ; offset of end of emulator register area

             dw      ? dup(?)    ; reserved
```

Comments

The **BASstk**, **CURstk**, and **LIMstk** fields specify the offsets from the start of the **SaveEmArea** member into the emulator's register area. If **BASstk** and **CURstk** have the same value, the stack is empty. Each of the emulator's registers is 12 bytes long and has the form shown in the following illustration.



The mantissa contains the leading 1 before the decimal point in the high-order bit of the most significant byte (msb). The exponent is not biased, that is, it is a signed integer. The following illustration shows the flag and tag bytes.



See Also **__Win87EmRestore, __Win87EmSave**

Screen Saver Library

Chapter 14

The Microsoft Windows operating system provides special applications called screen savers that start when the mouse and keyboard have been idle for a period of time. Screen savers exist for two main reasons:

- To avoid phosphor burn caused by static images on a screen
- To conceal sensitive information left on a screen

Clearing a screen addresses both goals, but screen savers are not restricted to this simple use. They can also display animated sequences such as a fish tank or fireworks. Animated sequences avoid phosphor burn by continually changing the image.

Windows provides a screen saver application that monitors the mouse and keyboard and starts the screen saver after a period of inactivity. The Desktop section of Windows Control Panel makes it possible for users to select from a series of screen savers, specify how much time should elapse before the screen saver is started, configure screen savers, and preview screen savers.

This chapter describes how to create a custom screen saver and add it to the library of screen savers users can select by using Control Panel.

14.1 About Screen Savers

Screen savers are Windows applications that contain specific variable declarations, exported functions, and resource definitions. The static-link library SCRNSAVE.LIB contains the **WinMain** function and other startup code required for a screen saver. To create a screen saver, you create a source module containing specific function and variable definitions and link it with SCRNSAVE.LIB. Your screen saver module is responsible only for configuring itself and for providing visual effects.

Screen savers are loaded automatically when Windows starts or when a user activates the screen saver feature by using Control Panel. Windows monitors keystrokes and mouse movements and starts the screen saver after a period of inactivity specified by the user.

Windows does not start the screen saver if any of the following conditions exists:

- The active application is not a Windows application.
- A computer-based training (CBT) window is present.
- The active application returns a nonzero value in response to the WM_SYSCOMMAND message sent with the SC_SCREENSAVE identifier.

When your screen saver starts, the startup code in SCRNSAVE.LIB creates a full-screen window. The window class for the screen saver window is declared as follows:

```
WNDCLASS cls;

cls.hCursor         = NULL;
cls.hIcon           = LoadIcon(hInst, MAKEINTATOM(ID_APP));
cls.lpszMenuName    = NULL;
cls.lpszClassName   = "WindowsScreenSaverClass";
cls.hbrBackground   = GetStockObject(BLACK_BRUSH);
cls.hInstance       = hInst;
cls.style           = CS_VREDRAW | CS_HREDRAW
                      | CS_SAVEBITS | CS_DBLCLKS;
cls.lpfnWndProc     = ScreenSaverProc;
cls.cbWndExtra      = 0;
cls.cbClsExtra      = 0;
```

Your source module provides the **ScreenSaverProc** window procedure. Your resource-definition file supplies the icon identified by **ID_APP**. This icon is visible only when your screen saver is run as a stand-alone application. (To be run by Control Panel, a screen saver must have the .SCR filename extension; to be run as a stand-alone application, it must have the .EXE filename extension.)

14.2 Creating a Screen Saver

The SCRNSAVE.H header file defines the function prototypes for the screen saver functions in SCRNSAVE.LIB. You must include this header file in your source module.

You must also define the **idsAppName** string. The **idsAppName** string should contain a screen saver name of the form **Screen Saver.***Name*, where *Name* is a unique name for your screen saver. For example, a screen saver named Bouncer would include the following line in the **STRINGTABLE** statement in its resource-definition file:

```
STRINGTABLE PRELOAD
BEGIN
    idsAppName          "Screen Saver.Bouncer"
    .
    .  /* other strings */
    .
END
```

If your screen saver stores configuration information, it should use the **idsAppName** string as the application heading for the configuration block in the CONTROL.INI file. For more information about storing screen saver configuration information, see Section 14.2.2, "Providing a Configuration Routine."

Your application should declare the following global variables, which are defined in SCRNSAVE.LIB:

```
extern HANDLE hMainInstance;
extern HWND   hMainWindow;
```

The **hMainInstance** variable contains the instance handle of your application. The **hMainWindow** variable contains the window handle of the screen saver window.

14.2.1 Processing Screen Saver Messages

Your screen saver module must include a **ScreenSaverProc** window procedure to receive and process messages for the screen saver window. The **ScreenSaverProc** window procedure must pass unprocessed messages to the **DefScreenSaverProc** function rather than to the **DefWindowProc** function.

Your **ScreenSaverProc** window procedure can substitute its own actions for the message handling performed by **DefScreenSaverProc**. For information about how the **DefScreenSaverProc** function responds to key window messages, see Section 14.5, "Functions."

The **ScreenSaverProc** window procedure must be exported by including it in the **EXPORTS** section of your module-definition (.DEF) file.

14.2.2 Providing a Configuration Routine

When the user chooses the Setup button, Windows uses the **/c** or **-c** command-line option to start the screen saver. To start the screen saver without displaying the configuration dialog box, Windows uses the **/s** or **-s** command-line option. When no command-line option is used, Windows displays the configuration dialog box, just as if **/c** had been specified.

If your screen saver supports configuration by the user, your source module must provide the following functions and dialog box resource to handle configuration:

Name	Description
ScreenSaverConfigureDialog	Dialog box procedure for a configuration dialog box.
RegisterDialogClasses	Function that registers any special or non-standard window classes needed for a configuration dialog box.
DLG_SCRNSAVECONFIGURE	Dialog box template for a configuration dialog box.

When Windows starts your screen saver with the configuration option (**/c**), the **WinMain** function in SCRNSAVE.LIB calls the **RegisterDialogClasses** function and then displays the configuration dialog box.

Define the **ScreenSaverConfigureDialog** function as you would any dialog box procedure.

Your screen saver should save its configuration settings in the CONTROL.INI file. SCRNSAVE.LIB uses the application name stored in the **idsAppName STRINGTABLE** statement as the CONTROL.INI application heading. Your application can use the **LoadString** function to retrieve the name of the heading from CONTROL.INI and then use the **WritePrivateProfileString** and **Write-PrivateProfileInt** functions to store the configuration information.

The *hInst* parameter of the **RegisterDialogClasses** function contains the instance handle for the screen saver. This is the same value contained in the **hMain-Instance** global variable. If your configuration routine does not require any special window classes, your **RegisterDialogClasses** function can simply return TRUE.

14.2.3 Creating Module-Definition and Resource-Definition Files

Be sure to export the **ScreenSaverProc** function and, if it is present, the **Screen-SaverConfigureDialog** function. The **RegisterDialogClasses** function should not be exported.

The **DESCRIPTION** statement in your module-definition file must use the following format:

DESCRIPTION 'SCRNSAVE : *description*'

If your screen saver includes a configuration routine, you should include a dialog box template with the **DLG_SCRNSAVECONFIGURE** identifier.

14.3 Installing New Screen Savers

Control Panel searches the Windows startup directory for files with the .SCR extension when compiling the list of available screen savers. (Screen saver applica-tions are standard Windows executable files. Simply rename the compiled screen saver so that its extension is .SCR.)

14.4 A Sample Screen Saver

The remainder of this chapter discusses the implementation of a screen saver application.

14.4.1 General-Purpose Declarations

Screen savers must use the string identifier **idsAppName** to identify themselves for other routines in SCRNSAVE.LIB:

```
STRINGTABLE PRELOAD
BEGIN
    idsAppName          "Screen Saver.ScreenSaverName"
    .
    .  /* other strings */
    .
END
```

The **idsAppName** string contains the name of the screen saver. The name to the right of the period is a unique name for the screen saver. The screen saver application can retrieve this string by calling the **LoadString** function.

Screen savers must also declare the following external variables:

```
HINSTANCE hMainInstance;
HWND      hMainWindow;
```

These external variables are defined in SCRNSAVE.LIB. They contain handles to the application instance and main window.

14.4.2 Message Handling

The following **ScreenSaverProc** function processes the WM_CREATE, WM_TIMER, WM_DESTROY, and WM_ERASEBKGND messages before calling the **DefScreenSaverProc** function:

```
LONG FAR PASCAL ScreenSaverProc(hWnd, msg, wParam, lParam)
HWND hWnd;
WORD msg;
WORD wParam;
LONG lParam;
{
    RECT rc;
    static WORD wBottomCount;
    switch (msg)
   {
       case WM_CREATE: {

           HANDLE hResInfo;
           GetIniEntries();   /* load strings from STRINGTABLE */
           GetIniSettings();  /* load initialization settings  */

           /* Load DIB image. */

           hbmImage = LoadBitmap(hMainInstance, szDIBName);

           /* Create a timer to move the image. */

           wTimer = SetTimer(hWnd, ID_TIMER, wElapse, NULL);

           xPos = xPosInit;
           yPos = yPosInit;

           break;
       }

       case WM_TIMER:

           if (bPause && bBottom) {
               if (++wBottomCount == 10) {
                   wBottomCount = 0;
                   bBottom = FALSE;
               }
               break;
           }

           MoveImage(hWnd);   /* move the image slightly */

           break;

       case WM_DESTROY:

           if (hbmImage)
               DeleteObject(hbmImage);
           if (wTimer)
               KillTimer(hWnd, ID_TIMER);

           break;

       case WM_ERASEBKGND:
           GetClientRect(hWnd, &rc);
           FillRect((HDC) wParam, &rc,
               (HBRUSH) GetStockObject(BLACK_BRUSH));
           return 0L;
```

```
        default:
            break;
    }

    return DefScreenSaverProc(hWnd, msg, wParam, lParam);
}
```

If your window procedure traps the WM_DESTROY message, it must use one of the following methods to properly end the screen saver:

- After processing the message, pass it to the **DefScreenSaverProc** function.
- In the WM_DESTROY case of the message handler, call the **PostQuitMessage** function.

14.4.3 Configuration Dialog Box

A screen saver uses the **ScreenSaverConfigureDialog** function to process messages sent to the configuration dialog box. (A screen saver's resource-definition file includes the dialog box template.) The configuration dialog box is displayed when the user selects the Setup button from Desktop section of Control Panel.

The **ScreenSaverConfigureDialog** function saves its configuration information in the CONTROL.INI file. This configuration information is largely specific to the screen saver and may include such settings as speed, color, number of objects, and position.

The configuration information may also include password protection. When a screen saver is password protected, the user cannot deactivate it and return to the Windows session without typing the password in a dialog box. Adding password protection to a screen saver requires three dialog boxes: one for setting or changing the password, one for typing the password after the screen saver has been activated, and one for informing the user when the password is incorrect. These dialog boxes can be defined as follows:

```
#define         ID_OLDTEXT      100
#define         ID_NEWTEXT      101
#define         ID_AGAIN        102
#define         ID_PASSWORD     103
#define         ID_ETOLD        104
#define         ID_ETNEW        105
#define         ID_ETAGAIN      106
#define         ID_ETPASSWORD   107
#define         ID_ICON         108
#define         ID_PASSWORDHELP 109
```

```
#ifdef RC_INVOKED

DLG_CHANGEPASSWORD    DIALOG      8,16,174,79
FONT 8, "MS Sans Serif"
STYLE WS_POPUP | DS_MODALFRAME | WS_CAPTION | WS_SYSMENU
CAPTION "Change Password"
BEGIN
 LTEXT "&Old Password:", ID_OLDTEXT,       4, 3,80,14
 EDITTEXT  ID_ETOLD,                      84, 3,80,14, ES_PASSWORD
 LTEXT "&New Password:", ID_NEWTEXT,       4,21,80,14
 EDITTEXT  ID_ETNEW,                      84,21,80,14, ES_PASSWORD
 LTEXT "&Retype New Password:", ID_AGAIN,  4,39,80,14
 EDITTEXT  ID_ETAGAIN,                    84,39,80,14, ES_PASSWORD
 DEFPUSHBUTTON "OK", IDOK,                 4,59,40,14
 PUSHBUTTON "&Help", ID_PASSWORDHELP,             64,59,40,14
 PUSHBUTTON "Cancel", IDCANCEL,          124,59,40,14
END


DLG_ENTERPASSWORD     DIALOG      250,175,170,96
FONT 8, "MS Sans Serif"
STYLE WS_POPUP | DS_MODALFRAME | WS_CAPTION | WS_SYSMENU
CAPTION "<name of screen saver>"
BEGIN
 LTEXT "The screen saver you are using is password protected.
     You must type the screen saver password
     to turn off the screen saver.", -1, 31,3,140,40
 LTEXT "Password:", ID_PASSWORD,          31,45,40,14
 EDITTEXT ID_ETPASSWORD,                  71,45,80,14, ES_PASSWORD
 DEFPUSHBUTTON "OK", IDOK,                31,66,40,14
 PUSHBUTTON "Cancel", IDCANCEL,          111,66,40,14
 ICON "", ID_ICON,                         3, 3,32,32
END

DLG_INVALIDPASSWORD DIALOG  8,16,174, 79
FONT 8, "MS Sans Serif"
STYLE WS_POPUP | DS_MODALFRAME | WS_CAPTION | WS_SYSMENU
CAPTION "<name of screen saver>"
BEGIN
 ICON "", ID_ICON,                         3, 3, 0, 0
 LTEXT "Incorrect password;\n\nCheck your screen saver password,
     and try again.", -1, 40,3,130,40
 DEFPUSHBUTTON "OK", IDOK,               70,50,40,14
END
#endif
```

The preceding example wraps long lines in the DLG_ENTERPASSWORD and DLG_INVALIDPASSWORD dialog boxes. These lines should not wrap in your resource-definition file.

The **ScreenSaverConfigureDialog** function typically processes a message from a check box that specifies whether the screen saver is password protected and a message specifying that the user has chosen the button to set the password, as shown in the following example:

```
case ID_SETPASSWORD: {
    FARPROC fpDialog;

    if ((fpDialog = MakeProcInstance(DlgChangePassword,
            hMainInstance)) == NULL)
        return FALSE;
    DialogBox(hMainInstance, MAKEINTRESOURCE(DLG_CHANGEPASSWORD),
            hDlg, fpDialog);
    FreeProcInstance(fpDialog);
    SendMessage(hDlg, WM_NEXTDLGCTL, hIDOK, 1l);
    break;
}

case ID_PASSWORDPROTECTED:
    bPassword ^= 1;
    CheckDlgButton(hDlg, wParam, bPassword);
    EnableWindow(hSetPassword, bPassword);
    break;
```

The **DlgChangePassword** function displays the DLG_CHANGEPASSWORD dialog box.

14.4.4 Adding Help

The configuration and password dialog boxes for screen savers typically include a Help button. Screen saver applications can check for the Help-button identifier and call the **WinHelp** function in the same way Help is provided in other Windows applications. In addition, SCRNSAVE.LIB includes **HelpMessageFilterHook-Function**, which posts the MyHelpMessage message whenever the user presses the F1 key while using a screen saver dialog box. A screen saver can check for this message in the **ScreenSaverConfigureDialog** function, as follows:

```
switch (msg) {
    .
    . /* process messages */
    .

    default:
        if (msg==MyHelpMessage)
            DoLocalHelpFunc();
}
```

14.4.5 Exporting Functions

A typical module-definition file for a screen saver application might look like this:

```
NAME        BOUNCER

DESCRIPTION 'SCRNSAVE : Bounce a bitmap'

STUB        'WINSTUB.EXE'
EXETYPE     WINDOWS

CODE        MOVEABLE DISCARDABLE PRELOAD
DATA        MOVEABLE MULTIPLE PRELOAD

HEAPSIZE    1024
STACKSIZE   4096

EXPORTS
        ScreenSaverProc                  @1
        ScreenSaverConfigureDialog       @2
        DlgChangePassword                @3
        DlgGetPassword                   @4
        DlgInvalidPassword               @5
        HelpMessageFilterHookFunction    @6
```

The **ScreenSaverProc**, **ScreenSaverConfigureDialog**, **DlgChangePassword**, and **HelpMessageFilterHookFunction** functions have been discussed earlier in this chapter. The screen saver module typically does not make explicit calls to the **HelpMessageFilterHookFunction**, **DlgGetPassword**, or **DlgInvalidPassword** function.

14.5 Functions

This section describes the functions that applications can use to create a screen saver.

DefScreenSaverProc

3.1

```
#include <scrnsave.h>

LRESULT DefScreenSaverProc(hwnd, msg, wParam, lParam)
HWND hwnd;           /* handle of screen saver window    */
UINT msg;            /* message                          */
WPARAM wParam;       /* first message parameter          */
LPARAM lParam;       /* second message parameter         */
```

The **DefScreenSaverProc** function provides default processing for any messages that a screen saver application does not process. All window messages that are not explicitly processed by the screen saver application's **ScreenSaverProc** window procedure must be passed to the **DefScreenSaverProc** function.

Parameters

hwnd
Identifies the screen saver window.

msg
Specifies the message to be processed. The **DefScreenSaverProc** function responds to messages that affect screen saver operation as follows:

Message	Response
WM_ACTIVATE, WM_ACTIVATEAPP, WM_NCACTIVATE	Closes the screen saver if *wParam* is FALSE, unless the password option is enabled in the configuration dialog box. If the password option is enabled, these messages are ignored. A *wParam* value of FALSE indicates that the screen saver is losing the input focus. The screen saver is closed by sending a WM_CLOSE message.
WM_SETCURSOR	Removes the cursor from the screen by setting the cursor to NULL.
WM_LBUTTONDOWN, WM_RBUTTONDOWN, WM_MBUTTONDOWN, WM_KEYDOWN, WM_KEYUP, WM_MOUSEMOVE	Posts a WM_CLOSE message to close the screen saver window, unless the password option is enabled. If the password option is enabled, a WM_MOUSEMOVE message displays the dialog box created by the **DlgGetPassword** function.
WM_DESTROY	Calls the **PostQuitMessage** function to close the screen saver.

Message	Response
WM_SYSCOMMAND	Returns FALSE if the *wParam* parameter of the WM_SYSCOMMAND message is either SC_SCREENSAVE or SC_CLOSE.

If a screen saver application must perform a different action in response to any of these messages, the application's **ScreenSaverProc** window procedure should process the message and not call **DefScreenSaverProc** for that message.

wParam
Specifies 16 bits of additional message-dependent information.

lParam
Specifies 32 bits of additional message-dependent information.

Return Value

The return value specifies the result of the message processing and depends on the message sent.

Comments

A screen saver application's **ScreenSaverProc** window procedure should use **DefScreenSaverProc** in place of the **DefWindowProc** function. The **DefScreenSaverProc** function passes any messages that do not affect screen saver operation to **DefWindowProc**.

See Also

ScreenSaverProc

DlgChangePassword

3.1

#include <scrnsave.h>

```
BOOL DlgChangePassword(hDlg, message, wParam, lParam)
HWND hDlg;          /* handle of dialog box         */
UINT message;       /* message                      */
WPARAM wParam;      /* first message parameter      */
LPARAM lParam;      /* second message parameter     */
```

The **DlgChangePassword** function receives messages from a dialog box that changes the password for a screen saver.

Parameters

hDlg
Identifies the dialog box that changes the password for a screen saver.

message
Specifies the message.

wParam
Specifies 16 bits of additional message-dependent information.

lParam
Specifies 32 bits of additional message-dependent information.

Return Value

The return value is nonzero if the function is successful; otherwise, it is zero.

Comments

This function is called by the **ScreenSaverConfigureDialog** function to change the password for a screen saver. An application uses the **MakeProcInstance** function with **DlgChangePassword** to display a configuration dialog box.

A password applies to all screen savers using SCRNSAVE.LIB. Whether the password is enabled, however, is specific to a particular screen saver.

The dialog box template for the change password dialog box must use the DLG_CHANGEPASSWORD identifier (defined as 2000).

The **DlgChangePassword** function must be exported by including it in an **EXPORTS** statement in the application's module-definition (.DEF) file.

See Also

DlgGetPassword, **DlgInvalidPassword**, **ScreenSaverConfigureDialog**

DlgGetPassword

3.1

#include <scrnsave.h>

```
BOOL DlgGetPassword(hDlg, message, wParam, lParam)
HWND  hDlg;             /* handle of dialog box          */
UINT  message;          /* message                       */
WPARAM wParam;          /* first message parameter       */
LPARAM lParam;          /* second message parameter      */
```

The **DlgGetPassword** function receives messages from the dialog box that retrieves the user's password.

Parameters

hDlg
Identifies the dialog box that retrieves the user's password.

message
Specifies the message.

wParam
Specifies 16 bits of additional message-dependent information.

lParam
Specifies 32 bits of additional message-dependent information.

Return Value
The return value is nonzero if the function is successful; otherwise, it is zero.

Comments
The **DlgGetPassword** function is provided in SCRNSAVE.LIB. Most applications provide a dialog box template and export the function without explicitly calling it in their code. This reference information for **DlgGetPassword** is provided for applications that change the default behavior.

The **DlgGetPassword** function is called by the **DefScreenSaverProc** function to retrieve the password for a screen saver.

A password applies to all screen savers using SCRNSAVE.LIB. Whether the password is enabled, however, is specific to a particular screen saver.

The dialog box template for the dialog box that retrieves the user's password must use the DLG_ENTERPASSWORD identifier (defined as 2001).

The **DlgGetPassword** function must be exported by including it in an **EXPORTS** statement in the application's module-definition (.DEF) file.

See Also
DefScreenSaverProc, **DlgChangePassword**, **DlgInvalidPassword**

DlgInvalidPassword

3.1

#include <scrnsave.h>

```
BOOL DlgInvalidPassword(hDlg, message, wParam, lParam)
HWND hDlg;           /* handle of dialog box          */
UINT message;        /* message                       */
WPARAM wParam;       /* first message parameter       */
LPARAM lParam;       /* second message parameter      */
```

The **DlgInvalidPassword** function displays a dialog box warning that a user's password is invalid.

Parameters

hDlg
Identifies the dialog box that warns that a user's password is invalid.

message
Specifies the message.

wParam
Specifies 16 bits of additional message-dependent information.

lParam
Specifies 32 bits of additional message-dependent information.

Return Value

The return value is nonzero if the function is successful; otherwise, it is zero.

Comments

The **DlgInvalidPassword** function is provided in SCRNSAVE.LIB. Most applications provide a dialog box template and export the function without explicitly calling it in their code. This reference information for **DlgInvalidPassword** is provided for applications that change the default behavior.

DlgInvalidPassword is called during processing of the **DlgGetPassword** function when the user types an incorrect password.

A password applies to all screen savers using SCRNSAVE.LIB. Whether the password is enabled, however, is specific to a particular screen saver.

The dialog box template for the dialog box warning that the user's password is invalid must use the DLG_INVALIDPASSWORD identifier (defined as 2002).

The **DlgInvalidPassword** function must be exported by including it in an **EXPORTS** statement in the application's module-definition (.DEF) file.

See Also

DlgChangePassword, **DlgGetPassword**

HelpMessageFilterHookFunction

3.1

#include <scrnsave.h>

```
DWORD HelpMessageFilterHookFunction(nCode, wParam, lpMsg)
int nCode;           /* identifier of hook      */
WORD wParam;         /* virtual-key code        */
LPMSG lpMsg;         /* address of message      */
```

The **HelpMessageFilterHookFunction** function posts a message when the user presses the F1 key while using one of the screen saver dialog boxes.

Parameters

nCode
Specifies a code used by the Windows hook function (also called the message-filter function) to determine how to process the message.

wParam
Specifies the virtual-key code pressed by the user.

lpMsg
Points to a message identifying the key event.

Return Value

The return value is TRUE if the function posts a message. Otherwise, it specifies the result of the default message processing and is determined by the value of the *nCode* parameter.

Comments

The **HelpMessageFilterHookFunction** function is provided in SCRNSAVE.LIB. Most applications export the function and check for the help message registered by the library without explicitly calling the function in their code. This reference information for **HelpMessageFilterHookFunction** is provided for applications that change the default behavior.

The **HelpMessageFilterHookFunction** function posts a registered message called MyHelpMessage. An application should check for this message in its **ScreenSaverConfigureDialog** function.

The **HelpMessageFilterHookFunction** function must be exported by including it in an **EXPORTS** statement in the application's module-definition (.DEF) file.

See Also

ScreenSaverConfigureDialog

RegisterDialogClasses

3.1

#include <scrnsave.h>

BOOL RegisterDialogClasses(*hInst*)
HANDLE *hInst*;   /* handle of application instance   */

The **RegisterDialogClasses** function registers any special or nonstandard window classes needed by a screen saver application's configuration dialog box.

Parameters

hInst
Identifies an instance of the module that is registering the window classes.

Return Value

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

Comments

The **RegisterDialogClasses** function should not be exported. It is called by routines defined in the SCRNSAVE.LIB file.

If a screen saver does not register any special window classes for the configuration dialog box, the **RegisterDialogClasses** function can simply return a nonzero value.

See Also **ScreenSaverConfigureDialog**

ScreenSaverConfigureDialog

3.1

#include <scrnsave.h>

BOOL ScreenSaverConfigureDialog(*hdlg*, *wmsg*, *wParam*, *lParam*)
HWND *hdlg*; /* handle of dialog box */
UINT *wmsg*; /* message */
WPARAM *wParam*; /* first message parameter */
LPARAM *lParam*; /* second message parameter */

The **ScreenSaverConfigureDialog** function receives messages sent to a screen saver application's configuration dialog box. A screen saver application that supports user configuration must provide this function.

Parameters

hdlg
Identifies the configuration dialog box.

wmsg
Specifies the message.

wParam
Specifies 16 bits of additional message-dependent information.

lParam
Specifies 32 bits of additional message-dependent information.

Return Value

The return value is nonzero if the function processes the message or zero if it does not, except in response to a WM_INITDIALOG message. In response to a WM_INITDIALOG message, **ScreenSaverConfigureDialog** should return zero if it calls the **SetFocus** function to set the input focus to one of the controls in the dialog box. Otherwise, it should return nonzero, in which case the system sets the input focus to the first control in the dialog box that can be given the focus.

Comments

An application uses the **MakeProcInstance** function with **ScreenSaver-ConfigureDialog** to display a configuration dialog box.

The dialog box template for the configuration dialog box must have the **DLG_SCRNSAVECONFIGURE** identifier.

A screen saver application should save its configuration settings in the CONTROL.INI file.

The dialog box procedure is used only if the default window class (WC_DIALOG) is used for the dialog box. The default class is used if no explicit class is given in the dialog box template. Although the dialog box procedure is similar to a window procedure, it must not call the **DefWindowProc** function to process unwanted messages. Unwanted messages are processed internally by the default dialog box procedure.

The **ScreenSaverConfigureDialog** function must be exported by including it in an **EXPORTS** statement in the application's module-definition (.DEF) file.

See Also **MakeProcInstance**, **RegisterDialogClasses**

ScreenSaverProc

3.1

#include <scrnsave.h>

```
LRESULT ScreenSaverProc(hwnd, wmsg, wParam, lParam)
HWND hwnd;          /* handle of screen saver window     */
unsigned wmsg;      /* message                           */
UINT wParam;        /* first message parameter           */
LPARAM lParam;      /* second message parameter          */
```

The **ScreenSaverProc** function receives messages sent to a screen saver window.

Parameters

hwnd
Identifies the window.

wmsg
Specifies the message.

wParam
Specifies 16 bits of additional message-dependent information.

lParam
Specifies 32 bits of additional message-dependent information.

Return Value The return value is the result of the message processing. It depends on the message that is processed.

Comments

A screen saver application's **ScreenSaverProc** window procedure should use the **DefScreenSaverProc** function instead of the **DefWindowProc** function to provide default message processing. The **DefScreenSaverProc** function passes any messages that do not affect screen saver operations to **DefWindowProc**.

The **ScreenSaverProc** function must be exported by including it in an **EXPORTS** statement in the application's module-definition (.DEF) file.

See Also

DefScreenSaverProc

Application Notes

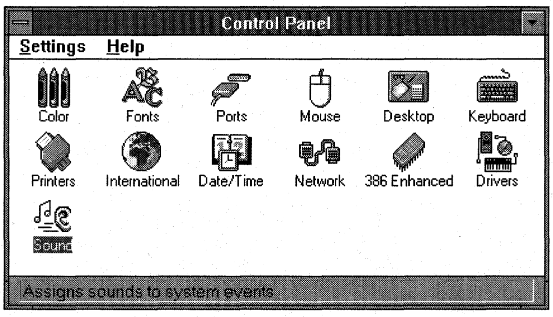
Part 3

Control Panel Applications

Chapter 15

This chapter describes Control Panel (CONTROL.EXE) for the Microsoft Windows operating system. It explains how to create a Control Panel application and then add the application to Control Panel.

Control Panel provides a window for running applications. These applications are used to configure the Windows environment. A number of standard applications are included with Windows. However, additional ones can be created and added to Control Panel. This capability is useful for modifying environmental factors unique to specific hardware and software. The following illustration shows the Control Panel window:



An application is contained in a dynamic-link library (DLL). A DLL can support more than one Control Panel application.

Control Panel loads Control Panel application libraries in this order:

1. The library containing the standard Control Panel applications
2. Libraries specified in the [MMCPL] section of the CONTROL.INI file
3. Libraries with the .CPL filename extension residing in the same directory as the CONTROL.EXE file
4. Libraries with the .CPL filename extension residing in the Windows SYSTEM directory

15.1 Starting a Control Panel Application

There are three ways to start a Control Panel application:

- The user can open Control Panel and start an application by double-clicking the application icon.
- The user or an application can open Control Panel by using a command-line argument that specifies the name of the application to start. When the Control Panel application closes, Control Panel automatically closes.

- An application can send a WM_CPL_LAUNCH message to Control Panel while Control Panel is running. When the Control Panel application closes, Control Panel sends back a WM_CPL_LAUNCHED confirmation message. For more information about these messages, see the *Microsoft Windows Programmer's Reference, Volume 3.*

The following example shows how an application can start Control Panel and the Printers application from the command line by using the **WinExec** function:

```
WinExec("control.exe printers", SW_SHOWNORMAL)
```

When Control Panel starts, it immediately displays the Printers application. After the Printers application finishes, Control Panel ends.

The following example shows a function that starts a Control Panel application by using the WM_CPL_LAUNCH message:

```
BOOL StartApplet(LPSTR lpszName, HWND hwndMine)
{
    HANDLE hAppletName;        /* global-object handle for app name   */
    HWND hwndCPL;              /* handle of Control Panel window      */
    LPSTR lpszAppletName;      /* name of the application             */
    BOOL  fStartedCPL = FALSE; /* application started by CONTROL.EXE? */

    /*
     * Allocate a global, sharable memory block to hold the
     * application-name string.
     */

    hAppletName = GlobalAlloc(GMEM_MOVEABLE | GMEM_NOT_BANKED,
        lstrlen(lpszName) + 1);
    if(hAppletName == (HANDLE) NULL)
        return FALSE;
    lpszAppletName = GlobalLock(hAppletName);
    lstrcpy(lpszAppletName, lpszName);
    GlobalUnlock(hAppletName);

    /*
     * Get the Control Panel window handle and start Control Panel, if
     * necessary.
     */

    if((hwndCPL = FindWindow((LPSTR) "CtlPanelClass",
            (LPSTR) "Control Panel")) == (HWND) NULL) {
        WinExec("control.exe", SW_SHOWNA);
        hwndCPL = FindWindow((LPSTR) "CtlPanelClass",
            (LPSTR) "Control Panel");
        if(!hwndCPL) {
            GlobalFree(hAppletName);
            return FALSE;
        }
        fStartedCPL = TRUE;
    }

    /* Start the application and end Control Panel, if started. */

    SendMessage(hwndCPL, WM_CPL_LAUNCH, (WPARAM) hwndMine,
        (LPARAM) lpszAppletName);
    if(fStartedCPL)
        SendMessage(hwndCPL, WM_CLOSE, 0, 0L);
    GlobalFree(hAppletName);
    return TRUE;
}
```

15.2 Creating a Control Panel Application

A Control Panel application must reside in a DLL that includes a standard entry-point function named **CPlApplet**. The application must include the CPL.H header file for the definition of the Control Panel messages. Control Panel communicates with the DLL by sending the following CPL messages to the **CPlApplet** function:

Message	Description
CPL_DBLCLK	Sent when the user double-clicks an application icon. In response to this message, the DLL should start its configuration process, usually displaying a dialog box.
CPL_EXIT	Sent after the last CPL_STOP message and immediately before Control Panel calls the **FreeLibrary** function for the DLL. In response to this message, the DLL should free any remaining memory and prepare to exit.
CPL_GETCOUNT	Sent after the CPL_INIT message, to prompt the DLL to return a number indicating how many applications it services.
CPL_INIT	Sent immediately after the DLL is loaded, to prompt the DLL to perform initialization procedures, including memory allocation.
CPL_INQUIRE	Sent after the CPL_GETCOUNT message, to prompt the DLL to provide information about each application. The handler for this message is a good place to include any initialization required by individual applications.

Message	Description
CPL_NEWINQUIRE	Sent to a Control Panel DLL to request information about an application that the DLL supports. The CPL_NEWINQUIRE message is the same as the CPL_INQUIRE message except that its second parameter (*lParam2*) is a pointer to a **NEW-CPLINFO** structure instead of a **CPLINFO** structure. New applications should use CPL_NEWINQUIRE instead of CPL_INQUIRE.
CPL_SELECT	Sent when the user selects an application icon.
CPL_STOP	Sent once for each application before Control Panel ends. In response to this message, the DLL should free any memory associated with the individual application for which the message is sent.

For more information about these messages, see the *Microsoft Windows Programmer's Reference, Volume 3.*

15.2.1 Creating the Entry-Point Function

Control Panel communicates with an application DLL through the **CPlApplet** function. Be sure to export this function by listing it in the **EXPORTS** statement of your module-definition (.DEF) file. The **CPlApplet** function handles the messages listed previously, performing three main tasks:

Task	Result
Initializing the application (CPL_INIT, CPL_INQUIRE)	Allocates any memory needed and gives Control Panel the information needed to display the application icon.
Running the application (CPL_DBLCLK)	Passes control to a dialog box and its associated message processor.
Closing the application (CPL_STOP, CPL_EXIT)	Frees any memory allocated and prepares to exit.

The **CPlApplet** function has the following format:

LONG CALLBACK* CPlApplet(*hwndCPL*, *iMessage*, *lParam1*, *lParam2*)

The *hwndCPL* parameter contains the handle of the Control Panel window. The *iMessage* parameter contains one of the CPL messages listed previously. The *lParam1* and *lParam2* parameters contain message-dependent values. For more information about the **CPlApplet** function, see the *Microsoft Windows Programmer's Reference, Volume 2.*

15.2.2 Initializing the Application

To initialize a Control Panel application, Control Panel sends the CPL_INIT message first to the **CPlApplet** function, which prompts the application DLL to perform initialization procedures. If initialization succeeds, **CPlApplet** returns nonzero.

If **CPlApplet** returns zero in response to the CPL_INIT message, Control Panel calls the **FreeLibrary** function and ends communication with the application DLL. This is the only way an application can notify Control Panel of initialization problems and prevent the application from being loaded.

If initialization is successful, Control Panel sends the CPL_GETCOUNT message. The **CPlApplet** function responds by returning the number of applications serviced by the application DLL. This number determines how many icons Control Panel displays for the DLL.

Once Control Panel finds out the number of applications serviced by the DLL, it sends the CPL_NEWINQUIRE message once for each application. The *lParam1* parameter specifies the application number, which is zero for the first application and CPL_GETCOUNT minus 1 for the last application.

Control Panel passes a far pointer to a **NEWCPLINFO** structure in the *lParam2* parameter. The **NEWCPLINFO** structure has the following form:

```
typedef struct tagNEWCPLINFO /* ncpli */
{
    DWORD dwSize;          /* length of structure, in bytes     */
    DWORD dwFlags;         /* setup flags                       */
    DWORD dwHelpContext;   /* help-context number               */
    LONG  lData;           /* application-defined data          */
    HICON hIcon;           /* handle of icon (owned by CPL.EXE) */
    char  szName[32];      /* short-name string                 */
    char  szInfo[64];      /* description string (status line)  */
    char  szHelpFile[128]; /* path to help file                 */
} NEWCPLINFO;
```

The **CPlApplet** function must fill in the **NEWCPLINFO** structure. The function must assign values for the **dwSize**, **hIcon**, **szName**, and **szInfo** members for the structure size, application icon, short name, and description. To add an accelerator key for the application, precede the selected accelerator character in the **szName** string with an ampersand. If the application DLL supports context-sensitive Help, the **CPlApplet** function should also assign the values for the **dwHelpContext** and **szHelpFile** members. The **lData** member can be used for application-defined data.

Note The CPL_NEWINQUIRE message and **NEWCPLINFO** structure replace the CPL_INQUIRE message and **CPLINFO** structure. The latter have been kept for backward compatibility with Windows version 3.0. If the application DLL does not respond to the CPL_NEWINQUIRE message, Control Panel sends it the CPL_INQUIRE message. Then the *lParam2* parameter points to a **CPLINFO** structure rather than to a **NEWCPLINFO** structure. For more information about these structures, see the *Microsoft Windows Programmer's Reference, Volume 3.*

15.2.3 Responding to User Actions

Control Panel sends the CPL_SELECT and CPL_DBLCLK messages when the user selects (single-clicks) or double-clicks an application icon. For each message, Control Panel passes the application number in *lParam1* and the **lData** value in *lParam2*.

Typically, an application DLL responds to the CPL_SELECT message by doing nothing. When it receives the CPL_DBLCLK message, it transfers control to the appropriate dialog box.

15.2.4 Exiting the Application and the DLL

Before exiting, Control Panel sends the CPL_STOP message once for each application in the DLL. The *lParam1* and *lParam2* parameters sent with the CPL_STOP message correspond to the application number and the **lData** value. After Control Panel sends the last CPL_STOP message, it sends a CPL_EXIT message and then calls the **FreeLibrary** function to free the DLL.

When the CPL_STOP and CPL_EXIT cases in the switch statement are executed, the DLL frees memory that it allocated. Typically, the DLL frees memory associated with individual applications when the CPL_STOP case is executed and frees any other allocated memory when the CPL_EXIT case is executed.

15.2.5 Example of a Control Panel Application

The following example shows the **CPlApplet** function for a DLL containing three Control Panel applications that set preferences for a component stereo system attached to the computer.

The example uses a programmer-defined StereoApplets array that contains three structures, each corresponding to one of the Control Panel applications. Each structure contains all the information required by the CPL_INQUIRE message, as well as the dialog box template and dialog box procedure required by the CPL_DBLCLK message. The following example fills the structures in the StereoApplets array:

```
#define NUM_APPLETS 3

typedef struct tagApplets
{
    int icon;         /* icon-resource identifier                  */
    int namestring;   /* name-string resource identifier           */
    int descstring;   /* description-string resource identifier    */
    int dlgtemplate;  /* dialog box template resource identifier */
    FARPROC dlgfn;    /* dialog box procedure                      */
} APPLETS;

APPLETS StereoApplets[NUM_APPLETS] =
{
    AMP_ICON, AMP_NAME, AMP_DESC, AMP_DLG, AmpDlgProc,
    TUNER_ICON, TUNER_NAME, TUNER_DESC, TUNER_DLG, TunerDlgProc,
    TAPE_ICON, TAPE_NAME, TAPE_DESC, TAPE_DLG, TapeDlgProc,
};
```

This code defines the **CPlApplet** function for the preceding example:

```
LONG FAR PASCAL CPlApplet(hwndCPL, iMessage, lParam1, lParam2)
HWND hwndCPL;          /* handle of Control Panel window */
unsigned int iMessage; /* message                        */
LONG lParam1;          /* first message parameter        */
LONG lParam2;          /* second message parameter       */
{
    int i;
    LPCPLINFO lpCPlInfo;

    i = (int) lParam1;

    switch (iMessage) {
        case CPL_INIT: /* first message, sent once */
            return ((LONG) TRUE);

        case CPL_GETCOUNT: /* second message, sent once */
            return (NUM_APPLETS);
            break;

        case CPL_INQUIRE: /* third message, sent once per app */
            lpCPlInfo = (LPCPLINFO)lParam2;

            lpCPlInfo->idIcon = StereoApplets[i].icon;
            lpCPlInfo->idName = StereoApplets[i].namestring;
            lpCPlInfo->idInfo = StereoApplets[i].descstring;
            lpCPlInfo->lData  = 0L;

            break;

        case CPL_SELECT: /* application selected */
            break;
            case CPL_DBLCLK: /* application double-clicked */
                DialogBox(hInstance,
                    MAKEINTRESOURCE(StereoApplets[i].dlgtemplate),
                    hwndCPL, (DLGPROC) StereoApplets[i].dlgfn);
                break;

            case CPL_STOP: /* sent once per app before CPL_EXIT */
                break;

            case CPL_EXIT: /* sent once before FreeLibrary called */
                break;

            default:
                break;
        }
        return (0L);
}
```

15.3 Installing a New Application

There are three ways to register an application DLL with Control Panel:

- List the DLL in the [MMCPL] section of the CONTROL.INI file. Use this method when the DLL is part of a system library and handles more than just messages from Control Panel. The following is a sample CONTROL.INI entry:

  ```
  [MMCPL]
  myapplets=mydll.dll
  ```

- Assign the DLL a .CPL filename extension and install it in the directory that contains the CONTROL.INI file.
- Assign the DLL a .CPL filename extension and install it in the Windows SYSTEM directory.

File Manager Extensions

Chapter 16

This chapter describes how to create and install extensions for File Manager in the Microsoft Windows operating system. A File Manager extension is a dynamic-link library (DLL) that adds a menu to File Manager.

File Manager maintains a list of extensions in an initialization file and loads the extensions when starting. An extension DLL contains an entry point that processes menu commands and notification messages sent by File Manager. Up to five extension DLLs can be installed at any one time.

16.1 Creating a File Manager Extension

A File Manager extension must reside in a DLL that includes a standard entry point, the **FMExtensionProc** function. It must include the WFEXT.H header file that defines File Manager messages and structures. File Manager communicates with the extension DLL by sending the following messages to the DLL's **FM-ExtensionProc** function:

Message	Meaning
1 through 99	User has selected an item from the extension-supplied menu. The value is the identifier of the selected menu item.
FMEVENT_INITMENU	User has selected the extension's menu. The extension should initialize items in the menu.
FMEVENT_LOAD	File Manager is loading the extension DLL and prompts the DLL for information about the menu that the DLL supplies.
FMEVENT_SELCHANGE	Selection in the File Manager directory window or Search Results window has changed.
FMEVENT_UNLOAD	Extension DLL is being unloaded.
FMEVENT_USER_REFRESH	User has chosen the Refresh command from the Window menu. The extension should update items in the menu, if necessary.

For more information about these messages, see the following section. For information about the **FMExtensionProc** function, see the *Microsoft Windows Programmer's Reference, Volume 2.*

16.2 Creating the Entry-Point Function

File Manager communicates with an extension DLL through the **FMExtensionProc** function. Be sure to export this function by listing it in an **EXPORTS** statement of your module-definition (.DEF) file. The **FMExtensionProc** function handles the messages listed in the previous section, performing the following tasks:

Task	Action
Initializing the extension (FMEVENT_LOAD)	Provides File Manager with the name and handle of the menu and saves the menu-item delta value.
Initializing the menu (FMEVENT_INITMENU)	Initializes all top-level menu items and the items in any submenus.
Processing menu selections	Carries out commands that the user chooses from the extension's menu.
Processing file selections (FMEVENT_SELCHANGE)	Queries File Manager for information about the file that the user has selected from the directory window or Search Results window. For information about using the FM_GETFILESEL message to retrieve information about a selected file, see the *Microsoft Windows Programmer's Reference, Volume 3*.
Updating items in the menu (FMEVENT_USER_REFRESH)	Modifies the menu as appropriate when the user chooses File Manager's Refresh command from the Window menu.
Quitting the extension DLL (FMEVENT_UNLOAD)	Frees any memory allocated and prepares to exit.

The **FMExtensionProc** function is defined as follows:

```
HMENU FAR PASCAL FMExtensionProc(hwnd, wMsg, lParam)
HWND hwnd;
WORD wMsg;
LONG lParam;
```

The *hwnd* parameter identifies the File Manager window. An extension should use this window handle to specify the parent window for any dialog boxes or message boxes it needs to display. It should also use this handle to send query messages to File Manager. The *wMsg* parameter contains one of the File Manager messages listed previously. The *lParam* parameter contains a message-dependent value. The return value from the **FMExtensionProc** function depends on the value of the *wMsg* parameter.

The menu added to File Manager may be a hierarchical (cascaded) menu and may contain grayed, disabled, or checked menu items in addition to command items.

Menu items should be text only; owner-drawn menus and bitmap menus are not supported. Changing the check-mark bitmap is not supported.

Whenever File Manager calls the **FMExtensionProc** function, it waits to refresh its directory windows (for changes in the file system) until after the function returns. This allows the extension to perform large numbers of file operations without excessive repainting on the part of File Manager. The extension does not need to send the FM_REFRESH_WINDOWS message to notify File Manager to repaint its directory windows.

16.2.1 Loading the Extension

File Manager sends, first, the FMEVENT_LOAD message to the **FMExtensionProc** function. The *lParam* parameter that accompanies the FMEVENT_LOAD message points to an **FMS_LOAD** structure that File Manager uses to obtain information about the extension-supplied menu, including the menu name and menu handle. For detailed information about the **FMS_LOAD** structure, see the *Microsoft Windows Programmer's Reference, Volume 3.*

File Manager also uses the **FMS_LOAD** structure to pass the menu-item delta value to the extension. To avoid conflicts with its own menu-item identifiers, File Manager renumbers the menu-item identifiers in an extension-supplied menu by adding the delta value to each identifier. If an extension DLL needs to modify its menu after File Manager has loaded it, it must use the delta value. For example, to delete a menu item, the extension DLL finds the sum of the delta value and the menu item's identifier and then passes the sum as the *idItem* parameter to the **DeleteMenu** function.

16.2.2 Processing Menu Selections

The menu resource that you define for your extension's menu must use menu-item identifiers in the range 1 through 99. When the user selects an item, the extension receives a command notification, which is the actual identifier of the selected item as defined in the resource-definition file (which has the .RC filename extension). The command notification is not the sum of the delta value and the identifier. An extension DLL's **FMExtensionProc** function carries out commands by processing command notifications.

16.2.3 Initializing the Extension Menu

Whenever the user selects the extension's main menu item from File Manager's menu bar, File Manager sends the FMEVENT_INITMENU message to the extension DLL. An extension can use this message to initialize its menu items. For example, an extension can add check marks, disable items, or gray items during this message.

When the user selects submenus within the extension's menu, File Manager does not send the FMEVENT_INITMENU message. An extension DLL must initialize all items at the same time, including those in submenus.

16.2.4 Updating the Extension Menu

When the user chooses the Refresh command from the Window menu, File Manager sends an FMEVENT_USER_REFRESH message to an extension DLL. An extension can use this opportunity to update its menu items.

16.2.5 Processing File Selections

When the user selects a filename in the directory window or in the Search Results window, File Manager sends the FMEVENT_SELCHANGE message to an extension DLL. An extension can use this opportunity to send a query message to File Manager to obtain more information about the user's selection. For more information, see Section 16.4, "Extension Messages."

Because the user can change the selection often, the extension should return promptly after processing the FMEVENT_SELCHANGE message to avoid slowing the user's selection process.

16.2.6 Quitting the Extension DLL

When File Manager quits, it sends the FMEVENT_UNLOAD message to each extension DLL and then calls the **FreeLibrary** function to free the DLLs. Each DLL should free any memory that it has allocated.

16.3 Installing Extensions

File Manager installs extensions that have settings in the [AddOns] section of the WINFILE.INI initialization file. Each setting contains an entry and a value. An entry consists of a string that represents the name of an extension. The value assigned to the entry consists of a string that specifies the path to the extension DLL. An application can use the **WritePrivateProfileString** function to add a setting to WINFILE.INI. The following example shows a setting in WINFILE.INI:

```
[AddOns]
My File Manager Extension=c:\win\system\rfmine.dll
```

File Manager does not display an error message if it cannot find an extension DLL, so an extension DLL can be deleted in order to uninstall it. Even so, an application that installs an extension DLL should provide an uninstall option to remove the extension's setting from the WINFILE.INI file.

16.4 Extension Messages

An extension can send the following window messages to retrieve relevant information from File Manager. File Manager is only guaranteed to respond correctly to messages sent from the **FMExtensionProc** function. For more information about these messages, see the *Microsoft Windows Programmer's Reference, Volume 3.*

Message	Description
FM_GETDRIVEINFO	File Manager returns drive information from the active window. An extension provides a pointer to an **FMS_GETDRIVEINFO** structure; File Manager fills the structure with drive information.
FM_GETFILESEL	File Manager returns information about a selected file from the active File Manager window (either the directory window or the Search Results window). An extension provides a pointer to an **FMS_GETFILESEL** structure; File Manager fills the structure with file information.
FM_GETFILESELLFN	Same as the FM_GETFILESEL message except that the selected file may have a long filename.
FM_GETFOCUS	File Manager returns a value that identifies the type of window with input focus.
FM_GETSELCOUNT	File Manager returns the count of selected files in the directory and Search Results windows.
FM_GETSELCOUNTLFN	Same as the FM_GETSELCOUNT message except that the count includes files with long filenames.
FM_REFRESH_WINDOWS	File Manager repaints either its active window or all of its windows. This message is similar to File Manager's Refresh command on the Window menu.

Message	Description
FM_RELOAD_EXTENSIONS	File Manager reloads all extensions. First File Manager unloads all extensions, sending an FMEVENT_UNLOAD message to each extension. Then it reloads the extensions, sending an FMEVENT_LOAD message to each extension. The FM_RELOAD_EXTENSIONS message allows an extension to uninstall itself by removing its setting from the WINFILE.INI file; this action causes File Manager to reload the remaining extensions. Other applications (for example, installation programs) can also post this message by calling the **PostMessage** function.

16.5 File Manager Extension Example

The following example shows the **FMExtensionProc** function for a sample extension DLL. It demonstrates how an extension processes the menu commands and notification messages sent by File Manager.

```
HINSTANCE hinst;
HMENU hmenu;
WORD wMenuDelta;
BOOL fMultiple = FALSE;
BOOL fLFN = FALSE;

DWORD FAR PASCAL FMExtensionProc(hwnd, wMsg, lParam)
HWND hwnd;
WORD wMsg;
LONG lParam;
{
    char szBuf[200];
    int count;

    switch (wMsg) {
        case FMEVENT_LOAD:

            #define lpload  ((LPFMS_LOAD)lParam)

            /* Save the menu-item delta value. */

            wMenuDelta = lpload->wMenuDelta;

            /* Fill the FMS_LOAD structure. */

            lpload->dwSize = sizeof(FMS_LOAD);
            lstrcpy(lpload->szMenuName, "&Extension");
        /* Return the handle of the menu. */

        return (DWORD) (lpload->hMenu = LoadMenu(hinst,
            MAKEINTRESOURCE(MYMENU)));
        break;

    case FMEVENT_UNLOAD:

        /* Perform any cleanup procedures here. */

        break;

    case FMEVENT_INITMENU:

        /* Copy the menu-item delta value and menu handle. */

        wMenuDelta = LOWORD(lParam);
        hmenu = (HMENU) HIWORD(lParam);

        /*
         * Add check marks to menu items as appropriate. Add menu-
         * item delta values to menu-item identifiers to specify the
         * menu items to check.
         */

        CheckMenuItem(hmenu, wMenuDelta + MULTIPLE,
            fMultiple ? MF_BYCOMMAND | MF_CHECKED :
                        MF_BYCOMMAND | MF_UNCHECKED);
        CheckMenuItem(hmenu, wMenuDelta + LFN,
            fLFN ? MF_BYCOMMAND | MF_CHECKED :
                   MF_BYCOMMAND | MF_UNCHECKED);
        break;

    case FMEVENT_USER_REFRESH:
        MessageBox(hwnd, "User refresh event", "Hey!", MB_OK);
        break;

    case FMEVENT_SELCHANGE:
        OutputDebugString("Sel change\r\n");
        break;

    /*
     * The following messages are generated when the user chooses
     * items from the extension menu.
     */

    case GETFOCUS:
        wsprintf(szBuf, "Focus %d", (int)SendMessage(hwnd,
            FM_GETFOCUS, 0, 0L));
        MessageBox(hwnd, szBuf, "Focus", MB_OK);
        break;
```

```
case GETCOUNT:
        count = (int)SendMessage(hwnd,
            fLFN ? FM_GETSELCOUNTLFN : FM_GETSELCOUNT, 0, 0L);

        wsprintf(szBuf, "%d files selected", count);
        MessageBox(hwnd, szBuf, "Selection Count", MB_OK);
        break;

    case GETFILE:
    {
        FMS_GETFILESEL file;

        count = (int) SendMessage(hwnd,
            fLFN ? FM_GETSELCOUNTLFN : FM_GETSELCOUNT,
            FMFOCUS_DIR, 0L);

        while (count >= 1) {

            /* Selection indices are zero-based (0 is first). */

            count--;
            SendMessage(hwnd, FM_GETFILESEL, count,
                (LONG) (LPFMS_GETFILESEL)&file);
            OemToAnsi(file.szName, file.szName);
            wsprintf(szBuf, "file %s\nSize %ld",
                (LPSTR)file.szName, file.dwSize);
            MessageBox(hwnd, szBuf, "File Information", MB_OK);

            if (!fMultiple)
                    break;
        }
        break;
    }

    case GETDRIVE:
    {
        FMS_GETDRIVEINFO drive;

        SendMessage(hwnd, FM_GETDRIVEINFO, 0,
            (LONG) (LPFMS_GETDRIVEINFO)&drive);

        OemToAnsi(drive.szVolume, drive.szVolume);
        OemToAnsi(drive.szShare, drive.szShare);

        wsprintf(szBuf,
    "%s\nFree Space %ld\nTotal Space %ld\nVolume %s\nShare %s",
            (LPSTR) drive.szPath, drive.dwFreeSpace,
            drive.dwTotalSpace, (LPSTR) drive.szVolume,
            (LPSTR) drive.szShare);
        MessageBox(hwnd, szBuf, "Drive Info", MB_OK);
        break;
    }
```

```
        case LFN:
            fLFN = !fLFN;
            break;

        case MULTIPLE:
            fMultiple = !fMultiple;
            break;

        case REFRESH:
        case REFRESHALL:
            SendMessage(hwnd, FM_REFRESH_WINDOWS,
                wMsg == REFRESHALL, 0L);
            break;

        case RELOAD:
            PostMessage(hwnd, FM_RELOAD_EXTENSIONS, 0, 0L);
            break;
    }
    return NULL;
}
```

16.6 Adding the Undelete Command

File Manager supports a hook for adding an Undelete command to the File menu (below the Delete command). If an undelete dynamic-link library is specified in the WINFILE.INI file, File Manager adds the Undelete command to the File menu when it starts. When the user chooses the Undelete command, File Manager calls the DLL.

The [settings] section of the WINFILE.INI file should include a reference to the undelete DLL, as follows:

```
[settings]
UNDELETE.DLL=C:\MYDIR\OTHER.DLL
```

An undelete DLL must include a standard entry point, the **UndeleteFile** function. This function must be exported by specifying the name of the function in the **EXPORTS** statement of the DLL's module-definition (.DEF) file.

The **UndeleteFile** function is defined as follows:

```
int FAR PASCAL UndeleteFile(hwndParent, lpszDir)
HWND hwndParent;
LPSTR lpszDir;
```

The *hwndParent* parameter identifies the parent window for any dialog boxes that the DLL creates. The *lpszDir* parameter specifies the initial directory to be used (for example, C:\TEMP). For more information about the **UndeleteFile** function, see the *Microsoft Windows Programmer's Reference, Volume 2.*

Shell Dynamic Data Exchange Interface

Chapter 17

This chapter describes the dynamic data exchange (DDE) interface of Windows Program Manager (PROGMAN.EXE). Program Manager is an application that lets users group, start, and otherwise control other applications for the Microsoft Windows operating system. Program Manager starts automatically when the user starts Windows and continues to run as long as Windows is in use. Upon starting, Program Manager displays one or more windows within its main window. Each window contains icons that correspond to logically related Windows applications. For example, the Main window contains an icon for the File Manager, Control Panel, Print Manager, Clipboard, MS-DOS Prompt, and Windows Setup applications.

The following topics are related to the information in this chapter:

- Atoms
- Dynamic data exchange (DDE)
- Registration database

17.1 PROGMAN.INI File

When Program Manager starts, it searches its initialization file for a list of group files. The windows that appear in Program Manager's main window correspond to group files. From the user's perspective, a group file is a collection of icons that represent logically related applications, but from the programmer's perspective, a group file is actually a collection of data. This data includes the color information for the icons (their AND and XOR masks), an offset to the resource header for each icon, the ideal resolution for displaying each icon, the name of the executable file that contains the application, and so on. For a description of the group file format, see the *Microsoft Windows Programmer's Reference, Volume 4.*

Group files are identified in the Program Manager initialization file. This initialization file, PROGMAN.INI, has the following form:

```
[Settings]
Window=64 48 576 384 1
Order= 3 4 5 6 8 7 2 1 9
AutoArrange=1
SaveSettings=1
MinOnRun=1
Startup=
display.drv=v776816.drv
```

```
[Groups]
Group1=C:\WINDOWS\MAIN.GRP
Group2=C:\WINDOWS\ACCESSOR.GRP
Group3=C:\WINDOWS\GAMES.GRP
Group4=C:\WINDOWS\STARTUP.GRP
Group5=C:\WINDOWS\LZEXPAND.GRP
Group6=C:\WINDOWS\COMDLG.GRP
Group7=C:\WINDOWS\GDI.GRP
Group8=C:\WINDOWS\WINPROJ.GRP
Group9=C:\WINDOWS\MICROSOF.GRP

[Restrictions]
NoRun=1
NoClose=1
NoSaveSettings=0
NoFileMenu=0
EditLevel=3
```

The following three sections describe the contents of the PROGMAN.INI file.

17.1.1 Settings Section

The first section of the initialization file, [Settings], controls attributes of the Program Manager environment. The following entries appear in the [Settings] section:

Entry	Meaning
Window=	Specifies the location and dimensions of Program Manager's main window.
Order=	Specifies the order in which the groups listed in the [Groups] section appear in Program Manager's main window.
AutoArrange=	Specifies whether Program Manager should automatically arrange icons within groups.
SaveSettings=	Specifies whether to save the position of Program Manager's main window when exiting Program Manager.
MinOnRun=	Specifies whether to minimize Program Manager when an application is started.
Startup=	Specifies the name of the startup group. Program Manager automatically starts the applications in the startup group whenever it starts. If the startup group has a name other than "Startup", that name must be specified by the Startup= entry.
display.drv=	Specifies the display driver that was in use when Program Manager last ended. When Program Manager starts, it compares this value to the string in the SYSTEM.INI file. If they are different, Program Manager reextracts the application icons.

17.1.2 Groups Section

The second section of the initialization file, [Groups], identifies the names of the group files for which Program Manager should display unique windows or icons. The groups must be numbered, but they need not be listed in any particular order. Program Manager never changes the number of an existing group, so if an application other than Program Manager constructs a PROGMAN.INI file, it can assign meaningful numbers to groups, if necessary.

17.1.3 Restrictions Section

The third section of the initialization file, [Restrictions], disables some capabilities of the Program Manager environment. The following entries can appear in the [Restrictions] section:

Entry	Meaning
NoRun=	Specifies whether to disable the Run command on the File menu. If this entry is set to 1, the command is disabled. If this entry is set to 0, the Run command is enabled. The default is 0 (enabled) if no value is specified.
NoClose=	Specifies whether to prevent the user from exiting Program Manager through the File menu, the System menu, the ALT+F4 accelerator, or the Task List. If this entry is set to 1, exiting is prevented. If this entry is set to 0, exiting is allowed. The default is 0 (allowing exiting) if no value is specified.
NoSaveSettings=	Specifies whether to disable the Save Settings on Exit command on the Options menu. If this entry is set to 1, the Save Settings on Exit command is disabled. If this entry is set to 0, the command is enabled. The default is 0 (enabled) if no value is specified.
NoFileMenu=	Specifies whether to disable the File menu and all of its commands. If this entry is set to 1, the File menu is disabled. If this entry is set to 0, the menu is enabled. The default setting is 0 (enabled) if no value is specified.
EditLevel=	Controls the extent to which the user can modify read-write groups. (Shared, read-only groups cannot be modified.) This entry may be set to one of the following values:

Value	Meaning
0	Allows any modifications to the group. This is the default.
1	Prevents the user from creating, deleting, or renaming groups.

Entry	Meaning *(continued)*
	Value — **Meaning**
	2 — Prevents the user from creating, deleting, or renaming groups and from creating or deleting items in a group.
	3 — Prevents the user from creating, deleting, or renaming groups; from creating or deleting items in a group; and from changing command lines for items in a group.
	4 — Prevents the user from changing any property of an item in a group; from creating, deleting, or renaming groups; from creating or deleting items in a group; and from changing command lines for items in a group.

Setting NoRun to 1 and EditLevel to 3 prevents a user from using Program Manager to run any applications that are not already in a program group.

17.2 Command-String Interface

Program Manager has a DDE command-string interface that allows other applications to create, display, delete and reload groups; add items to groups; replace items in groups; delete items from groups; and to close Program Manager. The following commands perform these actions:

AddItem
CreateGroup
DeleteGroup
DeleteItem (Windows version 3.1 only)
ExitProgman
Reload (Windows 3.1 only)
ReplaceItem (Windows 3.1 only)
ShowGroup

The setup program for an application can use these commands, for example, to instruct Program Manager to install the application's icon in a group.

Multiple commands may be concatenated; each command must be contained in square brackets, and parameters must be contained in parentheses and separated by commas. Quotation marks must be used to delimit arguments that contain spaces, brackets, or parentheses. For example, the following set of commands adds WINAPP.EXE to the Windows Applications group:

```
[CreateGroup(Windows Applications)]
[ShowGroup(1)]
[AddItem(winapp.exe,Win App,winapp.exe,2)]
```

To use these commands, an application must first initiate a conversation with Program Manager. The application and topic names for the conversation are both

PROGMAN. Then the application sends the WM_DDE_EXECUTE message, specifying the appropriate command and its parameters.

Note The user can configure Windows to use a shell other than Program Manager as the default. As a result, you should not design an application assuming that Program Manager will be available for a DDE conversation.

The following sections describe Program Manager DDE command strings in detail. In the syntax blocks in the following sections, brackets enclose optional arguments.

17.2.1 CreateGroup

The syntax for the **CreateGroup** command has this form:

CreateGroup(*GroupName*[,*GroupPath*])

The **CreateGroup** command instructs Program Manager to create a new group or activate the window of an existing group.

Following are the parameters for this command:

GroupName
Identifies the group to be created. This parameter is a string. If a group already exists with the name specified by *GroupName*, **CreateGroup** activates the group window.

GroupPath
Specifies the path of the group file. If your application does not supply this parameter, Windows uses a default filename for the group in the Windows directory.

17.2.2 ShowGroup

The syntax for the **ShowGroup** command has this form:

ShowGroup(*GroupName*,*ShowCommand*)

The **ShowGroup** command instructs Program Manager to minimize, maximize, or restore the window of an existing group.

Following are the parameters for this command:

GroupName
Identifies the group window to be minimized, maximized, or restored.

ShowCommand
Specifies the action that Program Manager is to perform on the group window. This parameter is an integer. It must have one of the following values:

Value	Meaning
1	Activates and displays the group window. If the window is minimized or maximized, Windows restores it to its original size and position.
2	Activates the group window and displays it as an icon.
3	Activates the group window and displays it as a maximized window.
4	Displays the group window in its most recent size and position. The window that is currently active remains active.
5	Activates the group window and displays it in its current size and position.
6	Minimizes the group window.
7	Displays the group window as an icon. The window that is currently active remains active.
8	Displays the group window in its current state. The window that is currently active remains active.

17.2.3 DeleteGroup

The syntax for the **DeleteGroup** command has this form:

DeleteGroup(*GroupName*)

The **DeleteGroup** command instructs Program Manager to delete an existing group.

Following is the parameter for this command:

GroupName
Identifies the group to be deleted.

17.2.4 Reload

The syntax for the **Reload** command has this form:

Reload(*GroupName*)

The **ReloadGroup** command instructs Program Manager to remove and reload an existing group. An application that modifies group files can use this command to cause Program Manager to update the groups when it has finished making modifications.

Following is the parameter for this command:

GroupName
Identifies the group to be removed and reloaded. If the *GroupName* parameter is not specified, Program Manager unloads all groups and reloads the [Group] section of PROGMAN.INI. The [Settings] and [Restrictions] sections are not reread.

17.2.5 AddItem

The syntax for the **AddItem** command has this form:

AddItem(*CmdLine*[,
Name[,*IconPath*[,*IconIndex*[,*xPos, yPos*[,*DefDir*[,
HotKey,[,*fMinimize*]]]]]]])

The **AddItem** command instructs Program Manager to add an icon to an existing group.

Following are the parameters for this command:

CmdLine
Specifies the full command line required to execute the application. This parameter is a string. At a minimum, this string is the name of the executable file for the application. It can also include the full path of the application and any parameters required by the application.

Name
Specifies the title that is displayed below the icon in the group window.

IconPath
Identifies the filename for the icon to be displayed in the group window. This parameter is a string. This file can be either a Windows executable file or an icon file. If the *IconPath* parameter is not specified, Program Manager uses the first icon in the file specified by the *CmdLine* parameter if that file is an executable file. If *CmdLine* specifies an associated file, Program Manager uses the first icon of the associated executable file. The association is taken from the registration database. (For more information about the registration database, see Chapter 7, "Shell Library.") If *CmdLine* specifies neither an executable file nor an associated executable file, Program Manager uses a default icon.

IconIndex
Specifies the index of the icon in the file identified by the *IconPath* parameter. The *IconIndex* parameter is an integer. PROGMAN.EXE contains five built-in icons that can be used for non-Windows programs.

xPos
Specifies the horizontal position of the icon in the group window. This parameter is an integer. You must use both the *xPos* and *yPos* parameters to specify the position of the icon. If you do not specify the position, Program Manager places the icon in the next available space.

yPos
Specifies the vertical position of the icon in the group window. This parameter is an integer. You must use both the *xPos* and *yPos* parameters to specify the position of the icon. If you do not specify the position, Program Manager places the icon in the next available space.

DefDir
Specifies the name of the default (or working) directory. This parameter is a string.

HotKey
Identifies a hot (or shortcut) key that is specified by the user.

fMinimize
Specifies whether an application window should be minimized when it is first displayed.

17.2.6 ReplaceItem

The syntax for the **ReplaceItem** command has this form:

ReplaceItem(*ItemName*)

The **ReplaceItem** command instructs Program Manager to delete an item and record the position of the deleted item. Program Manager will add a new item (specified by the next **AddItem** command) at this recorded position.

Following is the parameter for this command:

ItemName
Specifies the item to be deleted. Its position is recorded by Program Manager.

17.2.7 DeleteItem

The syntax for the **DeleteItem** command has this form:

DeleteItem(*ItemName*)

The **DeleteItem** command instructs Program Manager to delete an item from the currently active group.

Following is the parameter for this command:

ItemName
Specifies the item to be deleted from the currently active group.

17.2.8 ExitProgman

The syntax for the **ExitProgman** command has this form:

ExitProgman(*bSaveGroups*)

If Program Manager was started by another application, the **ExitProgman** command instructs Program Manager to exit and, optionally, save its group information.

Following is the parameter for this command:

bSaveGroups
Specifies a Boolean value that, if nonzero, causes Program Manager to save its group information before closing. If *bSaveGroups* is zero, Program Manager does not save its group information.

17.3 Requesting Group Information

Program Manager can provide information about its groups to an application. Applications can request this information from Program Manager by using the PROGMAN topic.

An application can obtain a list of Program Manager groups by issuing a request for the Group item. Program Manager provides the list in CF_TEXT format. The list consists of group-name strings separated by carriage returns.

An application can use a group name as an item name to request information about the group. Program Manager provides this information in CF_TEXT format. The fields of group information are separated by commas. The first line of the information contains the group name (in quotation marks), the path of the group file, and the number of items in the group. Each subsequent line contains information about an item in the group, including the command line (in quotation marks), the default directory, the icon path, the position in the group, the icon index, the shortcut key (in numeric form), and the minimize flag.

International Applications

Chapter 18

The Microsoft Windows operating system provides means for making applications country- and language-independent. This chapter describes how to design Windows applications so that they can be readily adapted to international markets. The following topics are related to the information in this chapter:

- File version library
- Resources and Resource Compiler (RC)
- Initialization files

18.1 Creating an International Application

To reach worldwide audiences, you need to design Windows applications so that they can be marketed in more than one country and modified for new markets. International applications must be country- and language-independent and easy to localize.

A Windows application, regardless of the language used in its interface, should be able to handle data from different countries and in different languages. For example, a database developed primarily for the English-speaking market should accept French and German input. The application should also support different currency symbols and date and time formats. Furthermore, it should permit complex operations, such as sorting, in any language selected by the user.

A Windows application should be developed so that localization can be easily accomplished. Localization is the process of adapting an application for a market other than the one for which it was originally designed. Adapting an application involves translating the product, adding new features when required, and modifying the product to meet local needs.

18.2 Achieving Country and Language Independence

Windows provides resources for writing applications that are country- and language-independent. These resources consist of international information stored in the WIN.INI file and in certain Windows functions. By using the resources described in this section, you can correctly produce international applications.

18.2.1 International Information in WIN.INI

The [Intl] section of the WIN.INI file contains the current country settings for Windows. The user can modify these settings through Control Panel. An application has access to the current country settings through the **GetProfileInt** and **GetProfileString** functions and can modify them through the **WriteProfileString** function. An application should read the required country settings at startup and should monitor the WM_WININICHANGE message to update its country settings in case the country settings in WIN.INI have changed.

Following are the country settings stored in WIN.INI:

iCountry
Country code. This value is based on the telephone country code. The only exception is Canada, which has 2 instead of 1 (1 is used by the United States). This setting controls country-dependent features not supported by Windows.

sCountry
String defining the selected country name.

sLanguage
National language code selected by the user. The International dialog box in Control Panel changes the language of the installed language-dependent module. Following are some of the language codes that Windows currently supports:

Code	Language
DAN	Danish
DEU	German
ENG	U.K. English
ENU	U.S. English
ESN	Modern Spanish
ESP	Castilian Spanish
FIN	Finnish
FRA	French
FRC	Canadian French
ISL	Icelandic
ITA	Italian
NLD	Dutch
NOR	Norwegian
PTG	Portuguese
SVE	Swedish

sList
List separator. This character separates elements in a list. The list separator must be different from the decimal separator to avoid conflicts with lists of numbers.

iMeasure
Measurement system selected by the user, where 0 equals metric and 1 equals English. This setting controls measurement-dependent features of an application.

iTime
Time format. This setting defines the time format: 12 hours or 24 hours, where 0 equals the 12-hour clock and 1 equals the 24-hour clock.

sTime
Time separator. This character is displayed between hours and minutes and between minutes and seconds.

s1159
Trailing string (A.M., for example) used in some countries for times between 00:00 and 11:59.

s2359
Trailing string (P.M., for example) for times between 12:00 and 23:59 when in 12-hour clock format or trailing string (GMT, for example) for any time when in 24-hour clock format.

iTLZero
Value specifying whether the hours displayed should have a leading zero, where 0 equals no leading zero (9:15, for example) and 1 equals a leading zero (09:15, for example).

iDate
Date format. Kept for compatibility with Windows 2*.x*. The values for this setting are: 0 equals Month-Day-Year, 1 equals Day-Month-Year, and 2 equals Year-Month-Day. The **sShortDate** setting should be used instead.

sDate
Date separator. Kept for compatibility with Windows 2*.x*. The **sShortDate** setting should be used instead.

sShortDate
Date picture of the short date format. The **sShortDate** setting accepts only the values m, mm, d, dd, yy and yyyy. For information about these values and the format of date pictures, see the **sLongDate** setting.

sLongDate
Date picture of the long date format, which is similar to the **sShortDate** setting, except it can also contain strings. Following are formats for different month (m), day (d), and year (y) values:

Value	Format
m	1–12
mm	01–12
mmm	Jan-Dec
mmmm	January-December
d	1–31
dd	01–31

Value	Format
ddd	Mon-Sun
dddd	Monday-Sunday
yy	00–99
yyyy	1900–2040

Following are examples of different date pictures:

Date picture	Example
d mmmm, yyyy	9 January, 1989
dddd, mmmm d, yyyy	Friday, February 7, 1992
m/d/yy	3/18/89
dd-mm-yyyy	18-03-1989
d "of" mmmm, yyyy	9 of January, 1992

sCurrency

Currency symbol of a given country. Use of this setting requires care. If the currency symbol is changed through Control Panel, do not make global replacements of currency amounts in your application. Once the user has entered an amount using a particular currency, that currency should stay the same. This setting also requires special attention when files are shared among users or applications.

iCurrency

Currency format. The values for this setting are as follows:

Value	Meaning
0	Currency symbol prefix with no separation ($1, for example)
1	Currency symbol suffix with no separation (1$, for example)
2	Currency symbol prefix with one character separation ($ 1, for example)
3	Currency symbol suffix with one character separation (1 $, for example)

iCurrDigits

Number of digits used for the fractional part of a currency amount.

iNegCurr

Negative currency format. The values for this setting are:

Value	Negative format
0	($1)
1	–$1
2	$–1
3	$1–
4	(1$)

Value	Negative format
5	–1$
6	1–$
7	1$–
8	–1 $
9	–$ 1
10	$ 1–

Note The dollar symbol represents any currency symbol defined by the **sCurrency** setting.

sThousand
Symbol used to separate thousands in numbers with more than three digits.

sDecimal
Character used to separate the integer part from the fractional part of a number.

iDigits
Value defining the number of decimal digits that should be used in a number.

iLzero
Value specifying whether a decimal value less than 1.0 (and greater than –1.0) should contain a leading zero, as follows:

Value	Meaning
0	Do not use a leading zero (.7, for example).
1	Use a leading zero (0.7, for example).

18.2.2 International Information in Windows Functions

Windows includes provisions for specifying a national language. Language, in conjunction with the specification of a country, allows Windows to describe more precisely the characteristics of a given geographical location (for example, Swiss-German as opposed to Swiss-French). The following Windows functions behave differently depending on the language that is selected:

AnsiLower
AnsiLowerBuff
AnsiNext
AnsiPrev
AnsiUpper
AnsiUpperBuff
IsCharAlpha
IsCharAlphaNumeric
IsCharLower
IsCharUpper
lstrcmp
lstrcmpi

18.2.2.1 Comparing and Sorting Strings

The **lstrcmp** and **lstrcmpi** functions allow applications to compare and sort strings based on the language specified by the user. These functions take into account different alphabetic orderings, diacritical marks, and special cases that require character compression or expansion. Note that the **lstrcmp** and **lstrcmpi** functions do not act the same way as the C run-time functions **strcmp** and **strcmpi.**

The comparison done by **lstrcmp** and **lstrcmpi** is based on a primary value and a secondary value (see the following illustration). Each character has a primary and a secondary value. For example, in the following matrix, the letter *d* has a primary value of 4 and a secondary value of 2.

Secondary values

Primary values	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	A	À	Á	Â	Ã	Ä	Å	a	à	á	â	ã	ä	å
2	B	b												
3	C	Ç	c	ç										
4	D	d												
5	E	È	É	Ê	Ë	e	è	é	ê	ë				
6	F	f												

When performing the comparison of two strings, the primary value takes precedence over the secondary value. That is, the secondary value is ignored unless a comparison based on primary value shows the strings as equivalent.

The following examples show the effect of primary and secondary values on string comparisons:

Comparison	Result
A = A	Primary values equal
A < a	Primary values equal, secondary values unequal (A < a)
Ab < ab	Primary values equal, secondary values unequal (A < a)
ab < Ac	Primary values unequal (b < c)

The **lstrcmpi** function ignores the effect of case in determining secondary value. That is, when **lstrcmpi** is called to compare AB and ab, the two strings are equivalent. However, **lstrcmpi** does not ignore diacritical marks, so Ab precedes äb regardless of whether the comparison is performed by the **lstrcmp** or **lstrcmpi** function.

When strings of different lengths are compared, length takes precedence over secondary values. That is, the shorter string always precedes the longer string as long as the primary values in the shorter string equal the primary values for equivalent characters in the longer string. For example, ab precedes ABC, but ABC precedes AD.

Depending on the language module installed, some characters are treated differently. For example, if the German language module is installed, the ß character expands to *ss*. If the Spanish language module is installed, the characters *ch* are treated as a single character that sorts between *c* and *d*.

18.2.2.2 Case Conversions

Use of the case conversion functions, **AnsiLower**, **AnsiLowerBuff**, **AnsiUpper**, and **AnsiUpperBuff**, varies depending on the language module installed. The **IsCharAlpha**, **IsCharAlphaNumeric**, **IsCharLower** and **IsCharUpper** functions are also language-dependent. Different languages treat case conversions differently.

Note Do not use the C-language case-conversion functions; they do not handle characters with values greater than 128 properly.

18.2.2.3 Handling Character Sets

If you are writing international Windows applications, you will handle different character sets. It is especially important in this case to understand the difference between the Windows and OEM character sets.

The Windows character set is essentially equivalent to the ANSI character set.

The OEM character set is defined by the Windows operating system as the character set used by MS-DOS. The term OEM does not refer to a specific character set; instead, it refers to any of the different character sets (code pages) that can be installed and used by MS-DOS.

Because Windows runs on top of MS-DOS, there must be a layer between Windows and MS-DOS that performs translations between Windows and OEM characters. When Windows is first installed, the Windows Setup program looks at the character set that has been installed by MS-DOS and then installs the correct translation tables and Windows OEM fonts.

Windows applications should use the Windows **AnsiToOem** and **OemToAnsi** functions when transferring information to and from MS-DOS. Also, applications should use the correct character set when creating filenames. For more information about handling filenames, see the following section.

There is no one-to-one mapping between the Windows and OEM character sets. Applying the **AnsiToOem** function and then the **OemToAnsi** function to a given string does not always result in the original string.

Because the Windows and OEM character sets are 8-bit character sets, always use unsigned char values instead of signed char values. Bugs that result from using signed char values are very hard to track.

18.2.2.4 Handling Filenames

Applications do file handling differently depending on factors such as speed, size, and programming style. This section describes the most common methods for handling filenames.

The easiest way of handling filenames in Windows is to use the Windows character set for all filenames and to use the **_lcreat**, **_lopen**, and **OpenFile** functions to deal with differences between the MS-DOS and the OEM character sets.

Another way to handle filenames is to use the **OpenFile** function to obtain a full path, by using the **szPathName** member from the **OFSTRUCT** structure. The **szPathName** member contains characters from the OEM character set and must first be converted to the Windows character set before it is used as a parameter for the **OpenFile** function, for other Windows functions, or in a dialog box.

The following example shows this conversion:

```
if (OpenFile("myfile.txt", &of, OF_EXISTS) == -1) {
    OemToAnsi(of.szPathName, szAnsiPath);
    OpenFile(szAnsiPath, &of, OF_CREATE);
}
```

The third, and maybe most complicated, way of handling files is to call MS-DOS directly (by using the **DOS3Call** function or an Interrupt 21h instruction). You must ensure that your application always passes OEM characters to MS-DOS.

Differences between the Windows and OEM character sets complicate the handling of filenames. Problems can occur when applications try to create filenames using the Windows character set that have no equivalent characters in the OEM set. For example, the character Ê does not exist in code page 437 (437 is the standard U.S. extended ASCII character set). If the application tries to save the file named Ê.TXT, Windows converts Ê.TXT into E.TXT (by using the **AnsiToOem** function) and then passes it to MS-DOS.

You can prevent confusion about filenames by using the ES_OEMCONVERT and CBS_OEMCONVERT control styles. These styles (the first for edit controls and the second for combo boxes) read the user's input and convert the typed character to a valid character (one that exists in the OEM character set). This way, the user sees on the screen the actual filename that will be stored at the MS-DOS level.

18.2.2.5 Handling the Keyboard

The most important keyboard issue for international applications is the use of the VK_OEM keys for user input because the locations of these keys change depending on the keyboard layout chosen by the user.

The **VkKeyScan** function is used to translate characters from the Windows character set into a virtual-key code plus a shift state. This function can be also used when one application has to send text to another application by simulating keyboard input.

Some other useful keyboard functions are the following:

Function	Purpose
ToAscii	Converts a virtual-key code plus a shift state to a character in the Windows character set. This function is the opposite of the **VkKeyScan** function.
GetKeyNameText	Retrieves a string that contains the name of a key (the SHIFT key or the ENTER key, for example). The string is in the language associated with the keyboard. For example, for a French keyboard layout the names of the keys are in French.
GetKBCodePage	Returns the code page (OEM character set) that was running at the MS-DOS level at the time Windows was installed. Note that there is no real relationship between the keyboard and the code page installed.

To type characters that are not on your keyboard, use the ALT key and the numeric keypad. For characters in the Windows character set, hold down ALT and then, using the numeric keypad, type 0 and the three-digit code of the character you want. For an OEM character, type the three-digit code for the character.

18.2.2.6 Handling Initialization Files

The WIN.INI and SYSTEM.INI files use the Windows character set. Usually, however, applications do not access SYSTEM.INI. For WIN.INI as well as for private initialization files, applications should use the following functions:

GetPrivateProfileInt	**GetProfileString**
GetPrivateProfileString	**WritePrivateProfileString**
GetProfileInt	**WriteProfileString**

The Windows character set should always be used with these functions.

The section names and setting names in WIN.INI and in private initialization files should be independent of the language of the application. Usually, all of these names remain in English. For example, in WIN.INI the section name [Desktop] and the setting name Wallpaper should always remain in English so that applications in different languages can access the same information.

18.2.3 International Uses of the File Version Library

If your application includes a Windows version resource, you can use the functions in the file version library (VER.DLL) in your installation program. A Windows version resource includes the language, code page, version number, and so on for a file. The functions in VER.DLL retrieve information from a file's version resource and install files based on this information. For example, if an installation program tries to replace an existing copy of an application with a new copy in a different language, the **VerInstallFile** function returns an error that indicates a language conflict. Then the installation program queries the user about whether to overwrite the old file, install the new copy in another location, or exit.

For more information about the contents of a version resource and about using version functions, see Chapter 11, "File Installation Library."

18.3 Achieving Easy Localization

Creating applications that are easy to localize is not difficult if you follow a few basic rules.

18.3.1 Isolation of Localizable Information

The most important rule for localization is to never mix functional code with strings, messages, or any other information that has to be modified to localize your application.

Hard-coded strings (strings mixed with functional code) make localization more difficult. In most Windows applications, all menus, strings and messages should be placed in the resource-definition (.RC) file. All the dialog box information should be placed in the dialog box script (.DLG) file. If you do this, you just need to run the Resource Compiler (RC) to obtain a new, localized version of the product instead of recompiling the executable file.

Strings that are not meant to be modified (filenames, WIN.INI setting names, and so on) can be placed in the .RC file, but the file should contain comments documenting that the names are permanent and should not be modified. It is a good idea also to mark what should be translated (explaining limitations, if any). The better you make the documentation, the easier the localization will be.

The .RC files and .DLG files should contain anything that can be a localization item. It is better to have extra information in these files than to have too little. In cases where an .RC or a .DLG file cannot be used, place all the information in a file, such as an include file, that is separate from any functional code.

18.3.2 Allocating Extra Space for Strings

Many languages are more verbose than English and require more space to hold strings or to display dialog boxes. There are cases, as with menus, where the space allocation is done dynamically, but in most cases the application has to provide the space. The following table shows the percentage of additional space that an application should allocate for non-English strings of various lengths.

Length in characters	Additional space required
1–10	200%
11–20	100%
21–30	80%
31–50	60%
51–70	40%
70+	30%

In the English version of your application, avoid creating dense menus where most of the available space (all except one line, for example) is used. Dialog boxes should be designed so that items can be moved freely, allowing reorganization of the contents as translation demands. Do not crowd status bars with information. Even abbreviations are often longer in other languages.

18.3.3 Handling Foreign Languages

Never make assumptions about language usage when dealing with foreign languages. The ordering of words can be different, and the number of words required is often greater than in English.

Keep in mind the following grammatical points when preparing an application for localizing:

- Avoid using the same word in more than one message. Some words, such as *none*, can have different translations (different gender and number) depending on the context.
- Do not create plurals of words by adding *s*. Keep two strings, one for the singular and one for the plural.
- Avoid using slang, abbreviations, or jargon, because they are difficult to translate.

Keep these syntactical considerations also in mind when localizing:

- Avoid parsing text to obtain information. Parsing normally assumes specific syntax.
- Do not create a long string from several short strings. The long string may not make sense in another language, because the order of parts of speech varies in different languages.

Incorporate graphic objects such as bitmaps, cursors, and icons with these considerations in mind:

- Avoid the use of embedded text in graphics. Text is difficult to modify when in graphical form. If you cannot avoid this, leave enough space for translation and try to create tools to simplify the modification.
- Look for graphic objects that represent international concepts, because graphic objects are also language dependent.

Keep in mind the following points when planning screen elements:

- Do not hard-code the position or size of any element on the screen, because an item changes position and size as it gets translated. In cases where you need to define the size or position of certain object, place this definition in the resource-definition (.RC) file.
- Use the **CreateWindow** function carefully. The *lpClassName* parameter should be constant and independent from localization, but the *lpWindowName* parameter, which is the string that appears in the title bar, should be localized. The string used for *lpWindowName* should be taken from the resources.

All messages should be self-contained, not dynamically assembled. In cases where messages have variables added to them at run time, do not make any assumptions about the position of the variable in the message. Handle variables in messages in the following manner:

1. Place the string containing the variable in the resource-definition (.RC) file:

   ```
   CannotOpen,       "The application could not open the file %s"
   ```

2. Use the **wsprintf** function to incorporate the variable into the string:

   ```
   LoadString(hInst, CannotOpen, lpFormat, MaxLen);
   wsprintf(FinalString, lpFormat, FileName);
   ```

Network Applications

Chapter 19

As local area networks (LANs) become increasingly common, application developers need to ensure that their applications run properly in a network environment. To do this, they should consider the behavior of applications shared by multiple users and the compatibility of applications that access network software directly with protected (standard or 386 enhanced) mode.

19.1 Sharing by Multiple Users

Many corporations choose to have their computer users share a single copy of an application that resides on a network server. The Microsoft Windows operating system, version 3.0 and later, can be run this way. The **/n** (network) option used in Windows Setup configures the user's system so that most Windows files are used directly off the network, but the user's personal files and configuration information are stored in a private Windows directory. (For more information about using a shared copy of Windows, see the *Microsoft Windows User's Guide*.)

If you intend to allow shared copies of your application, you must ensure that two users running the same application do not interfere with each other. The following sections present guidelines for preparing an application for network support.

19.1.1 Sharing Directories

Many applications store configuration files in the same directory as the executable file for the application. This method does not work for multiple users, however, because the application stores each user's information in the same directory, overwriting the other users' information in the process.

Instead of using configuration files, an application should use the Windows profile functions to store user-specific information in initialization (.INI) files. The profile functions create initialization files in a user's private Windows directory, unless the application specifies a different directory.

Windows profile functions, such as **WriteProfileString**, usually store profile and configuration information in .INI files. Profile functions fall into two categories: those that access WIN.INI and those that access another .INI file specified by the program.

The functions that access WIN.INI are **GetProfileString**, **GetProfileInt**, and **WriteProfileString**. Because each user has a unique copy of WIN.INI, these functions can be used safely, even when the application is being shared by more than one user.

The functions that access other .INI files are **GetPrivateProfileString**, **GetPrivateProfileInt**, and **WritePrivateProfileString**. These functions behave similarly to the functions that access WIN.INI, except that the application specifies the name of the private initialization file. When using these functions, you should specify the name of the file, but not a complete path (for example, MYAPP.INI instead of C:\MYAPP\MYAPP.INI). By default, the file will be located in the user's private Windows directory; specifying a full path could give multiple users access to the same file.

The exception to the preceding rule are initialization files that need to be shared by all users. Make sure that those files cannot be left in an inconsistent state if multiple users update them simultaneously.

For a full description of the profile functions, see the *Microsoft Windows Programmer's Reference, Volume 2.*

19.1.2 Sharing Temporary Storage

When creating temporary files, use the **GetTempFileName** function to determine a unique name and location for the file. This function ensures that temporary filenames do not conflict, even if multiple users share the same temporary storage directory.

19.1.3 Sharing Files

A network manages file sharing as if the SHARE utility were loaded. Each file that can be accessed on the network should use a sharing mode to ensure data integrity. Applications should also be designed to handle sharing violations.

A sharing violation occurs when one process (or machine) attempts to access a file after a different process has requested the server to block access to the file. If an application opens the file in compatibility mode, a sharing violation results in a critical error. Therefore, unless the application uses the **SetErrorMode** function to set the error mode so that it always fails, Windows displays the standard sharing violation message.

For more information on file sharing and record locking, see *The MS-DOS Encyclopedia* (Redmond, Washington: Microsoft Press, 1988).

19.1.4 Sharing Devices

Windows 3.1 includes three functions that an application can use to manage its network connections: **WNetAddConnection**, **WNetCancelConnection**, and **WNetGetConnection**. The **WNetAddConnection** function redirects a local device (either a disk drive or a printer port) to a shared device on a remote server.

The **WNetCancelConnection** function cancels a redirection to a shared device. The **WNetGetConnection** function returns the name of the network resource associated with a redirected local device. For more information about these network functions, see the *Microsoft Windows Programmer's Reference, Volume 2*.

19.2 Calling Network Software in Protected Mode

Windows applications running in protected mode require special support whenever they make a call to real-mode software. This includes calls to MS-DOS, the BIOS, or a network. Non-Windows applications running with Windows do not require this special support, however, because they always run in real or virtual-8086 mode.

Windows applications running in protected mode require application programming interface (API) mapping. If the arguments to the calling function include pointers to data, that data should be copied into the first 1 megabyte of address space so that the real-mode software can access it. The processor is then switched into real or virtual-8086 mode so that the real-mode software can process the function. Finally, when the function returns, any data it modified is copied back to the caller's protected-mode address.

Fortunately, most applications interact with the network only indirectly, by using MS-DOS functions to manipulate files on redirected drives or by using MS-DOS or BIOS functions to print to a remote printer using redirected printer ports. Windows applications can continue to perform these functions as usual, because Windows automatically maps standard MS-DOS and BIOS functions.

Some applications, however, need to use functions that are specific to a particular network or networking protocol. Some part of the software must map these functions, and, in some cases, this may require special procedures on the part of the programmer.

The remainder of this chapter describes programming considerations for designing Windows applications that use the following networking protocols and networks: Microsoft Networks and MS-DOS network functions, NetBIOS functions, Microsoft LAN Manager–based networks, Novell NetWare, Ungermann-Bass Net/One, and Banyan VINES.

19.2.1 Microsoft Networks and MS-DOS Network Functions

Many networks on the market today are based on the Microsoft Networks standard, also known as MS-NET. These networks support a set of standard MS-DOS functions that perform network activities, such as redirecting drive letters.

Current versions of Windows automatically handle these MS-DOS functions. However, in order to maintain compatibility with future Windows products, your application should not make MS-DOS calls by using Interrupt 21h. Instead, it should set up all the registers for Interrupt 21h and then make a far call to the Windows **DOS3Call** function.

For a full description of the **DOS3Call** function, see the *Microsoft Windows Programmer's Reference, Volume 2*. For more information about Microsoft Networks functions, see *The MS-DOS Encyclopedia.*

19.2.2 NetBIOS Functions

NetBIOS is the most widely used networking API. The functions in this API are normally called by using Interrupt 5Ch. Current versions of Windows handle most NetBIOS functions. However, in order to maintain compatibility with future Windows products, the application should not make the NetBIOS call by using Interrupt 5Ch. Instead, it should set up all the registers for Interrupt 5Ch and then make a far call to the Windows **NetBIOSCall** function.

Windows does not support the following rarely used NetBIOS functions:

Function number	Function name
71h	**Send.No.Ack**
72h	**Chain.Send.No.Ack**
73h	**Lan.Status.Alert**
78h	**Find.Name**
79h	**Trace**

For a full description of the **NetBIOSCall** function, see the *Microsoft Windows Programmer's Reference, Volume 2.*

19.2.3 LAN Manager Networks

Networks based on Microsoft LAN Manager can be installed in either basic or enhanced versions. All versions of LAN Manager support MS-NET and NetBIOS functions. However, if you are running the enhanced version of LAN Manager with the API option, your applications can also use a powerful set of networking functions.

Non-Windows applications can call networking functions by linking with DOSNET.LIB, a static-link library provided with the network software. Windows applications, however, must use two dynamic-link libraries (DLLs), NETAPI.DLL and PMSPL.DLL, distributed on every workstation with the enhanced version of

LAN Manager 2.0. (These DLLs do not run with LAN Manager 1.*x* or with the basic version of LAN Manager 2.0.)

For more details on writing Windows applications for LAN Manager, see the *Microsoft LAN Manager Programmer's Reference.*

19.2.4 Novell NetWare

Novell NetWare supports MS-NET and, optionally, NetBIOS functions, which are described earlier in this chapter. Novell NetWare also supports the NetWare and IPX/SPX APIs, both based on Interrupt 21h.

Windows applications cannot make NetWare calls by using Interrupt 21h directly, because this method is not supported in all Windows operating modes. Instead, the Interrupt 21h instruction should be replaced by a far call to the **NetWareRequest** function. This function is exported by name from the NetWare DLL and should be imported to the module-definition (.DEF) file as NetWare.NetWareRequest.

Windows applications cannot make IPX/SPX calls at this time, although Novell plans to make this support available in a future release. For more information, contact Novell product support.

19.2.5 Ungermann-Bass Net/One

Ungermann-Bass Net/One is based on the Microsoft Networks standard. It supports standard MS-NET functions and most NetBIOS functions described earlier in this chapter.

Net/One also supports private extensions to the NetBIOS function set (Interrupt 5Ch Functions 72h–7Dh). These functions are supported by Windows. You can call these functions as you would standard NetBIOS functions by making a far call to the **NetBIOSCall** function.

19.2.6 Banyan VINES

Banyan VINES supports the standard MS-NET functions and, optionally, NetBIOS functions. A toolkit is available for applications that write directly to the VINES API.

Windows applications can call the MS-NET and NetBIOS functions as previously described.

VINES version 4.0 does not support Windows applications that call the VINES API directly, but Banyan intends to make this support available in VINES 4.1. For more information, contact Banyan product support.

Windows Applications with MS-DOS Functions

Chapter 20

This chapter describes the support in the Microsoft Windows operating system version 3.0 and later for Windows and non-Windows applications using DOS Protected-Mode Interface (DPMI) version 1.0 functions, MS-DOS interrupts and functions in protected mode, and the NetBIOS in protected mode.

DPMI enables MS-DOS applications to access the extended memory of PC-architecture computers while maintaining system protection. It also defines a new interface, through Interrupt 31h, that protected-mode applications use for such tasks as allocating memory, modifying descriptors, and calling real-mode software.

According to the DPMI specification, the term real-mode software refers to code that runs in the low 1-megabyte address space and uses segment:offset addressing. With Windows 3.0 and later in protected mode, so-called real-mode software is actually run in virtual-8086 mode. However, because virtual-8086 mode is a close approximation of real mode, both are referred to as real mode in this chapter.

For more information about the DPMI specification, contact Intel Corporation product support, or submit a service request through Microsoft OnLine.

20.1 Using DOS Protected-Mode Interface Functions

Windows 3.0 and later in 386 enhanced mode supports DPMI version 1.0. Windows 3.0 and later in standard mode supports a subset of DPMI that enables applications to call terminate-and-stay-resident (TSR) programs and device drivers running in real (or virtual-8086) mode. To ease the porting of an application to other operating environments, all code that calls DPMI functions directly should reside in a dynamic-link library (DLL).

20.1.1 Windows Kernel

Windows applications should not use the MS-DOS memory management functions for DPMI. The Windows 3.0 and later kernel has two functions, **GlobalDOSAlloc** and **GlobalDOSFree**, that should be used by Windows applications and DLLs for allocating and freeing MS-DOS addressable memory.

Because the Windows kernel provides functions for allocating memory, manipulating descriptors, and locking memory, no DPMI functions other than the following are required for Windows applications:

Interrupt 21h function	Description
0200h	Get Real Mode Interrupt Vector
0201h	Set Real Mode Interrupt Vector
0300h	Simulate Real Mode Interrupt
0301h	Call Real Mode Procedure with Far Return Frame

Interrupt 21h function	Description
0302h	Call Real Mode Procedure with Interrupt Return Frame
0303h	Allocate Real Mode Callback Address
0304h	Free Real Mode Callback Address

Non-Windows applications running in 386 enhanced mode can use all DPMI functions, because those functions are not restricted by the kernel.

20.1.2 Other Application Programming Interfaces

In general, any software-interrupt function that passes parameters in the EAX, EBX, ECX, EDX, ESI, EDI, and EBP registers works as long as none of the registers contains a selector value. In other words, if a software-interrupt function is completely register-based without any pointers, segment registers, or stack parameters, that function should work with Windows running in protected mode.

More complex software interrupt functions require the calling function to use the DPMI translation functions.

20.2 Support for MS-DOS Interrupts

This section discusses support for MS-DOS interrupts and functions when Windows runs in protected mode with MS-DOS version 3.0 and later.

All MS-DOS interrupts and functions that are not mentioned in this section should work exactly as documented in *The MS-DOS Encyclopedia* (Redmond, Washington: Microsoft Press, 1988).

20.2.1 Unsupported MS-DOS Interrupts and Functions

The following MS-DOS interrupts are not supported in protected mode and will fail if called:

Interrupt	Description
20h	Terminate Program
25h	Absolute Disk Read
26h	Absolute Disk Write
27h	Terminate and Stay Resident

The following MS-DOS Interrupt 21h functions are also not supported in protected mode:

Function	Description
00h	Terminate Process
0Fh	Open File with FCB
10h	Close File with FCB
14h	Sequential Read
15h	Sequential Write
16h	Create File with FCB
21h	Random Read
22h	Random Write
23h	Get File Size
24h	Set Random Record Number
27h	Random Block Read
28h	Random Block Write

20.2.2 Partially Supported MS-DOS Interrupt 21h Functions

The following MS-DOS Interrupt 21h functions behave differently in protected mode than they do in real mode. To use these functions, an application might require additional code:

Function	Description
25h	Set Interrupt Vector
35h	Get Interrupt Vector
38h	Get/Set Current Country Information
4402–4405h	Send/Receive Control Data
440Ch	Generic IOCTL for Character Devices
6501–6506h	Get Extended Country Information

Functions 25h and 35h set and get the protected-mode interrupt vector. They can be used to hook hardware interrupts, such as the timer or keyboard interrupt, as well as to hook software interrupts. (Except for Interrupts 23h, 24h, and 1Ch, software interrupts that are issued in real mode are not passed to protected-mode interrupt handlers. However, all hardware interrupts are passed to protected-mode interrupt handlers before being passed to real mode).

Function 38h returns a 34-byte buffer containing a doubleword real-mode address. The address at offset 12h is used for case mapping. To call the case-mapping function, use the DPMI translation function to simulate a real-mode FAR call.

Functions 4402h, 4403h, 4404h, and 4405h are used to receive data from a device or send data to a device. Because it is not possible to break the transfers automatically into small pieces, the calling program should assume that a transfer of greater than 4K will fail unless the address of the buffer is in the low 1 megabyte.

Only certain extensions of Function 440Ch (Minor Codes 45h and 65h) are supported for protected mode. The extensions of Function 440Ch that are used for code-page switching (Minor Codes 4Ah, 4Ch, 4Dh, 6Ah, and 6Bh) are not supported for protected-mode programs. To use 440Ch to switch code pages, you must use the DPMI translation functions.

Functions 6501h, 6502h, 6503h, 6504h, 6505h, and 6506h are supported for protected-mode programs. However, all doubleword parameters returned will contain real-mode addresses (that is, the case-conversion procedure address and all the pointers to tables will contain real-mode segment:offset addresses). To call the case-conversion procedure in real mode, you must use the DPMI translation functions.

20.3 NetBIOS Support

Windows supports standard NetBIOS (Interrupt 5Ch) functions in protected mode. All network control blocks (NCBs) and buffers must reside in fixed memory that is page-locked. To ease the porting of the application to other operating systems, all code that calls NetBIOS functions directly should reside in a DLL.

For additional information on NetBIOS support in Windows network drivers, see the *Microsoft Windows Device Driver Adaptation Guide*. For more information about developing applications for networks, see Chapter 19, "Network Applications."

Windows Prologs and Epilogs

Chapter 21

This chapter describes the prolog and epilog used with far functions in applications and dynamic-link libraries (DLLs) for the Microsoft Windows operating system. Compiler vendors can use this information to enable their compilers to generate prolog and epilog code that is suitable for Windows.

In Windows version 3.0 and earlier, the prolog and epilog for far functions must include instructions to mark the stack frame, indicating that the frame belongs to a far function. This makes it possible for real-mode Windows to locate segment addresses on the stack and update those addresses when it moves or discards the corresponding segments. Marking stack frames for far functions also allows debugging applications, such as Microsoft CodeView® for Windows (CVW) and Microsoft Windows 80386 Debugger (WDEB386.EXE), to display meaningful information about the contents of an application's stack.

Marking stack frames for far functions is optional for Windows 3.1 applications. Old debugging applications that do not access TOOLHELP.DLL, however, still need marking. Debugging applications that use TOOLHELP.DLL do not require stack frames for far functions to be marked.

21.1 Data-Segment Initialization

The Windows prolog and epilog contain instructions that initialize the DS register, setting the register to the segment address of the application or DLL. Windows requires callback functions, such as window, dialog box, and enumeration procedures, to initialize the DS register whenever they are called by Windows or an application. This guarantees that the function accesses its own data segment rather than the data segment of the caller.

21.1.1 Exported Far Functions

The Windows prolog used with exported far functions, such as dialog box and enumeration procedures, ensures that the DS register receives the data segment address for the application when Windows or an application calls the exported function. In Windows version 3.0 and earlier, the prolog and epilog for exported far functions have the following form:

```
push    ds          ; put DS in AX, take 3 bytes to do it,
pop     ax          ;   so the code can be rewritten as
nop                 ;   MOV AX, IMM when appropriate

inc     bp          ; push odd BP to indicate this stack
push    bp          ;   frame corresponds to a far CALL

mov     bp, sp      ; set up BP to access arguments and
                    ;   local variables
```

```
push    ds          ; save DS
mov     ds, ax      ; set DS to proper data segment
sub     sp, const   ; allocate local storage (optional)

 ...

sub     bp, 2       ; restore registers
mov     sp, bp
pop     ds
pop     bp
dec     bp
retf
```

Because Windows 3.1 does not support real mode, the **inc bp** and **dec bp** instructions are not required. Also, a variety of other changes can be made to the prolog and epilog to improve speed and reduce the size of the code. If a far function is part of an application (not part of a DLL), the SS register is already the proper value for the DS register, so calling the **MakeProcInstance** function is not necessary. The prolog and epilog can be modified as follows:

```
push    bp          ; set up stack frame (optional)
mov     bp, sp

push    ds          ; save calling function's DS

push    ss          ; move SS to DS
pop     ds

 ...

pop     ds          ; restore registers
pop     bp

retf
```

An alternative form of the prolog and epilog for far functions follows:

```
push    bp          ; set up stack frame (optional)
mov     bp, sp

push    ds          ; save calling function's DS

mov     ax, ss      ; move SS to DS
mov     ds, ax

sub     sp, const   ; (optional) allocate local storage

 ...
```

```
mov     ds, [bp-2]  ; restore registers
leave

retf
```

Each of the variations of prolog and epilog code discussed previously works whether or not a far function is exported. The code can be called by an application or DLL as well as by the system.

If an application copies the contents of the SS register to the DS register, it doesn't need to call the **MakeProcInstance** function to obtain a procedure-instance address before calling an exported far function. Similarly, if a DLL moves the DGROUP data segment to the DS register through the AX register, the DLL doesn't need to call **MakeProcInstance** before calling an exported far function.

Although window procedures for an application require this same prolog, Windows loads the AX register before calling these procedures. An application, therefore, never needs to create a procedure-instance address for its window procedures.

21.1.2 Nonexported Far Functions

Although not required, nonexported far functions can also include prolog code that initializes the DS register. In this case, it is assumed that the function is never called by Windows or an application and that the DS register contains the correct segment address when the function is called. The prolog for a nonexported function has the following form:

```
mov     ax, ds      ; copy DS to AX
nop

push    bp          ; set up stack frame (optional)
mov     bp, sp

push    ds          ; save calling function's DS
mov     ds, ax      ; move same value back to DS

 ...

pop     ds          ; pop same value back to DS
pop     bp
retf
```

An alternative form of the prolog for a nonexported function follows:

```
push    ds          ; copy DS to AX
pop     ax
nop
```

```
push    bp          ; set up stack frame (optional)
mov     bp, sp

push    ds          ; save calling function's DS

mov     ds, ax      ; move same value back to DS

 ...

pop     ds          ; pop same value back to DS
pop     bp
retf
```

A compiler should not generate the preceding code by default because it reloads the DS register with the same value two times per far call. Loading segment registers is a slow operation in protected mode and should be avoided as much as possible.

21.1.3 Exported Far Functions in a Dynamic-Link Library

Exported far functions in DLLs also require a prolog. The prolog code in a DLL must generate a reference to the DGROUP data segment. The SS register cannot be used because execution occurs on the calling function's stack. Exported far functions cannot use this method because fixups to DGROUP are illegal for a multiple instance application.

The prolog and epilog for exported far functions in a DLL has the following form:

```
mov     ax, DGROUP      ; get DGROUP value

push    bp              ; set up stack frame (optional)
mov     bp, sp

push    ds              ; save calling function's DS
mov     ds, ax          ; move DGROUP to DS

 ...

pop     ds              ; restore registers
pop     bp

retf
```

Following is an alternative form of the prolog for exported far functions in a DLL:

```
mov     ax, DGROUP      ; get DGROUP value

push    bp              ; set up stack frame (optional)
mov     bp, sp
```

```
push    ds              ; save calling function's DS
mov     ds, ax          ; move DGROUP to DS

sub     sp, const       ; allocate local storage (optional)

 ...

mov     ds, [bp-2]      ; restore registers

leave
```

Windows inserts the current data segment address as the second operand (DGROUP) of the initial **mov** instruction.

21.2 Prologs in Real Mode

When Windows 3.0 and earlier is running in real mode, Windows must walk each application stack whenever it moves or discards segments. In particular, it must check each stack for any segment addresses that may have been affected by the segment operations.

To help Windows locate segment addresses associated with the stack frames of far functions, the Windows prolog increments the old frame pointer, contained in the BP register, before saving it on the stack. Because all stack offsets, including frame pointers, are expected to be word-aligned, incrementing the BP register gives Windows a quick way to locate all far function stack frames.

Windows only walks the stack in real mode. In protected mode, selector values do not change even though Windows may move and discard segments. Therefore, functions in protected mode do not need to increment the BP register when they save it. However, some debugging programs, such as CVW and WDEB386.EXE, use the incremented BP register to determine which stack frames correspond to far functions and give meaningless stack backtraces if the BP register is not incremented before it is saved.

21.3 Prologs in Protected Mode

Although exported functions in protected-mode, single-instance applications need to set the DS register, these functions do not require the exported prolog described in the previous section. Instead, they can use code similar to that generated by the **_loadds** keyword of the Microsoft C Optimizing Compiler (CL) to set the DS register.

The code generated by **_loadds** copies the data segment selector to the DS register whenever the function is called. Because a selector does not change value when the corresponding segment is moved, there is no need to set the AX register to the appropriate data segment address before calling the function (or to mark the stack frame). The function can, therefore, be called directly rather than through a procedure-instance address. The **_loadds** code has the following form:

```
push    bp
mov     bp,sp
push    ds
mov     ax, CONSTANT
mov     ds, ax
```

Functions that use the **_loadds** code can be used as callback functions. Because no prolog code is required, the functions do not need to be exported when used in an application. Functions in DLLs can also use the **_loadds** code. However, the functions must be exported to ensure that other applications can link dynamically to them.

In multiple-instance applications, the Windows prolog is needed only for far functions called by Windows. For these functions, procedure-instance addresses are required. The **_loadds** code cannot be used in multiple-instance applications. Instead, applications should copy the SS register to the DS register.

Windows Application Startup

Chapter 22

This chapter describes the startup requirements of applications for the Microsoft Windows operating system. It also discusses the steps needed to initialize an application before its entry-point function, **WinMain**, can be called.

Windows dynamic-link libraries (DLLs) also have startup requirements. For a complete description of the startup routines for those DLLs, see the *Microsoft Windows Guide to Programming*.

22.1 Startup Requirements

When Windows starts an application, it calls a startup routine supplied with the application rather than the application's **WinMain** function. The startup routine is responsible for initializing the application, calling **WinMain**, and exiting the application when **WinMain** returns control.

When Windows first calls the startup routine, the processor registers have the following values:

Register	Value
AX	Contains zero.
BX	Specifies the size, in bytes, of the stack.
CX	Specifies the size, in bytes, of the heap.
DI	Contains a handle identifying the new application instance.
SI	Contains a handle identifying the previous application instance.
BP	Contains zero.
ES	Contains the segment address of the program segment prefix (PSP).
DS	Contains the segment address of the automatic data segment for the application.
SS	Same as the DS register.
SP	Contains the offset to the first byte of the application stack.

To initialize and exit a Windows application, the startup routine must follow these steps:

1. Initialize the task by using the **InitTask** function. **InitTask** also returns values that the startup routine passes to the **WinMain** function.
2. Clear the event that started the task by calling the **WaitEvent** function.
3. Initialize the queue and support routines for the application by calling the **InitApp** function with the instance handle returned by the **InitTask** function.
4. Call the entry point for the application, the **WinMain** function.
5. Exit the application by calling the MS-DOS End Program function (Interrupt 21h Function 4Ch) when **WinMain** returns.

Although the startup routine is essentially the same for all Windows applications, a variety of startup routines may need to be developed to accommodate the different memory models and high-level language run-time libraries used by Windows applications. If a Windows application uses functions and variables provided by run-time libraries, the startup routine may need to be customized to initialize the library at the same time as the application. Customizing the startup routine for run-time library initialization is entirely dependent on the library and is, therefore, beyond the scope of this chapter.

22.2 Example of a Startup Routine

A startup routine initializes and exits a Windows application. The routine in the following example, the __astart function, shows the code needed for startup, which includes Cmacros defined in the CMACROS.INC header file. When assembled, this code is suitable for small-model Windows applications that do not use run-time libraries:

```
.xlist
memS = 1     ; small memory model
?DF = 1      ; Do not generate default segment definitions.
?PLM = 1;
?WIN = 1;
include cmacros.inc
.list

STACKSLOP = 256

createSeg   _TEXT,CODE,PARA,PUBLIC,CODE
createSeg NULL, NULL, PARA,PUBLIC,BEGDATA,DGROUP
createSeg _DATA,DATA, PARA,PUBLIC,DATA,   DGROUP
defGrp      DGROUP,DATA

assumes DS,DATA

sBegin      NULL
            DD  0
labelW      <PUBLIC,rsrvptrs>
maxRsrvPtrs = 5
            DW  maxRsrvPtrs
            DW  maxRsrvPtrs DUP (0)
sEnd        NULL

sBegin  DATA
staticW hPrev,0             ; Save WinMain parameters.
staticW hInstance,0
staticD lpszCmdline,0
staticW cmdShow,0
sEnd    DATA
```

```
externFP   <INITTASK>
externFP   <WAITEVENT>
externFP   <INITAPP>
externFP   <DOS3CALL>
externP    <WINMAIN>

sBegin  CODE
assumes CS,CODE

labelNP <PUBLIC,__astart>

        xor     bp,bp                   ; zero bp
        push    bp

        cCall   INITTASK                ; Initialize the task.
        or      ax,ax
        jz      noinit

        add     cx,STACKSLOP            ; Add in stack slop space.
        jc      noinit                  ; If overflow, return error.

        mov     hPrev,si
        mov     hInstance,di
        mov     word ptr lpszCmdline,bx
        mov     word ptr lpszCmdline+2,es
        mov     cmdShow,dx

        xor     ax,ax                   ; Clear initial event that
        cCall   WAITEVENT,<ax>          ;   started this task.
        cCall   INITAPP,<hInstance>     ; Initialize the queue.
        or      ax,ax
        jz      noinit

        cCall   WINMAIN,<hInstance,hPrev,lpszCmdline,cmdShow>
ix:
        mov     ah,4Ch
        cCall   DOS3CALL                ; Exit with return code from
app.
noinit:
        mov     al,0FFh                 ; Exit with error code.
        jmp short ix
sEnd     CODE

        end __astart                    ; start address
```

Windows requires the null segment (containing the rsrvptrs array), which is defined at the beginning of this sample. The **InitTask** function copies the top, minimum, and bottom address offsets of the stack into the third, fourth, and fifth elements of the rsrvptrs array. Applications can use these offsets to check the amount of space available on the stack. The debugging version of Windows also uses these offsets to check the stack. Applications must, therefore, not change these offsets, since doing so can cause a system debugging error (RIP).

22.3 Function Reference

This section provides information about the **InitApp**, **InitTask**, and **WaitEvent** functions mentioned earlier in this chapter.

InitApp

```
xternFP InitApp

push    hInstance       ; instance handle
call    InitApp

or      ax,ax           ; zero if error
jz      error_handler
```

The **InitApp** function creates the application queue and installs application-support routines, such as the signal procedure, version-specific resource loaders, and the divide-by-zero interrupt routine.

Parameters

hInstance
Identifies the task to be initialized. This parameter must have been previously supplied by Windows.

Return Value

This function returns a nonzero value in the AX register if successful. Otherwise, it returns zero in the AX register to indicate an error.

See Also

InitTask

InitTask

```
externFP InitTask

call    InitTask    ; Initialize a task.
```

The **InitTask** function initializes the task by setting registers, setting up the command line, and initializing the heap. This must be the first function called by the startup routine for the application.

Parameters

This function has no parameters.

Return Value

This function returns 1 in the AX register and fills the CX, DX, ES:BX, SI, and DI registers with information about the new task, if the function is successful. Otherwise, it returns zero in the AX register to indicate an error.

Comments

When the function is successful, other registers contain the following values:

Register	Value
CX	Contains the stack limit, in bytes. The startup routines should check the limit to ensure there is a minimum of 100 bytes in the stack.
DI	Contains the instance handle for the new task. The startup routine passes this address to the **WinMain** function.
DX	Contains an *nCmdShow* parameter. The startup routine passes this parameter to the **WinMain** function for use with the **CreateWindow** function.
ES	Contains the segment address of the program segment prefix (PSP) for the new task.
ES:BX	Contains the 32-bit address of the command line (MS-DOS format). The startup routine passes this address to the **WinMain** function.
SI	Contains the instance handle for the previous instance of the application, if any. The startup routine passes this address to the **WinMain** function.

The **InitTask** function also copies the top, minimum, and bottom address offsets of the stack to the 16 bytes of reserved memory at the beginning of the automatic data segment for the application. The reserved memory has the following format:

```
        DW  0
globalW oOldSP,0
globalW hOldSS,5
globalW pLocalHeap,0
globalW pAtomTable,0
globalW pStackTop,0
globalW pStackMin,0
globalW pStackBot,0
```

See Also

InitApp

WaitEvent

```
externFP WaitEvent

push    taskID       ; task identifier
call    WaitEvent

or      ax,ax
jnz     resched      ; nonzero if rescheduled
```

The **WaitEvent** function checks for a posted event and, if one is found, clears the event and returns control to the application. If no event is found, the function suspends execution of the application by calling the Windows scheduler.

Parameters

taskID
Identifies the task to check events for. If this parameter is zero, the function checks events for the current task.

Return Value

This function returns a nonzero value if the Windows scheduler has scheduled another application. Otherwise, it returns zero.

Video Techniques

Chapter 23

This chapter describes some techniques that can improve the video performance of applications for the Microsoft Windows operating system. These techniques include using an identity palette to speed up image drawing, accommodating differences in video adapters, and modifying device-independent bitmaps (DIBs) by using the DIB driver.

23.1 Using an Identity Palette

Windows reserves a group of system palette entries for a fixed number of colors. These colors, which are named system colors, are used for drawing screen elements such as scroll bars. Windows also uses the system colors as replacement color entries when inactive windows request more color entries than are available in the system palette. Windows places the system colors at the top and bottom of the system palette to ensure that logical operations (such as XOR) work correctly.

By arranging logical palettes the same way that Windows arranges the system palette, you can avoid unexpected color changes and improve the speed at which your application draws DIBs. To do this, you must create an identity palette, a logical palette that matches the system palette. To use identity palettes, however, you need to understand how Windows sets up the system palette.

23.1.1 Understanding the System Palette

When an application realizes a palette (that is, requests the palette be given specified colors), Windows adds the logical palette entries to the system palette. Windows always reserves system palette entries for the system colors. For example, a 256-color video graphics adapter (VGA) driver with 20 system colors allows an application to use a maximum of 236 system palette entries. If a logical palette contains more entries than can fit in the system palette (after the system colors are added), Windows truncates the palette, using only as many colors as it can fit without encroaching on the reserved system colors. You can force Windows to relinquish the system color entries (by using the **SetSystemPaletteUse** function), but by doing so you change the coloring of all Windows screen elements to black and white.

The maximum number of colors available to a foreground window equals the number of colors supported by the video driver minus the number of system reserved colors and the number of palette entries reserved by the application.

Windows places the system colors at the top and bottom of the system palette. For example, a 256-color VGA driver uses the top 10 and bottom 10 system palette entries for the system colors. If a logical palette does not contain the system colors or if the system colors appear in locations other than the default positions, Windows changes the ordering of the palette entries when your application realizes its palette. At this point, logical palette entry *n* does not necessarily match system palette entry *n*. When your application draws a bitmap to the device context, Windows must translate the bitmap palette indices to the new locations on the system palette. This translation step takes time.

The goal is to make the logical palette exactly match the system palette. By doing so, your images can be colored exactly as you expect. The video driver can also draw the images faster because the translation step is avoided.

23.1.2 Creating an Identity Palette

An identity palette is a logical palette that exactly matches the system palette and therefore has the same number of entries as the system palette and includes color entries for the system colors. The system colors appear at the top and bottom of the color table.

The Microsoft Windows Paintbrush application always saves bitmaps with an identity palette. To convert a bitmap palette to an identity palette, you can open the bitmap in Paintbrush and then save it.

23.2 Accommodating Different Video Adapters and Drivers

This section contains information on adapting your logical palette to a different display type.

23.2.1 Distinguishing Between Standard VGA and Super VGA

Most super VGA adapters are single-plane devices, which makes them well-suited for displaying DIBs. On a super VGA adapter, there is little speed difference between drawing DIBs and drawing device-dependent bitmaps—you can choose whichever format is more convenient for your application.

Standard VGA adapters have multiple planes and are not as well suited for displaying DIBs. It is faster to work with device-dependent bitmaps on standard VGA. To determine whether a standard VGA adapter is present, use the following code:

```
hDC = CreateDC("DISPLAY", NULL, NULL, NULL);

bIsMultiplane = (GetDeviceCaps(hDC, PLANES) > 1);

DeleteDC(hDC);
```

23.2.2 Adapting Identity Palettes to Different Display Adapters

Even if two display devices use the same number of system colors, you cannot assume that the red, green, and blue (RGB) values for the low-intensity colors match. One particular problem is the difference between super VGA and 8514 systems. Both provide 256 colors and use 20 system colors, but the low-intensity system color values for the VGA are different from those for the 8514. An identity palette created on a VGA system is not the same as an identity palette on an 8514 system.

If you create an identity palette on a VGA system and then display the DIB on an 8514 system, Windows recognizes the low-intensity colors in the logical palette as custom colors rather than system colors. It puts these colors in the custom-color section of the palette (in entries 10 through 245) and the 8514 system colors in the top and bottom of the system palette.

To avoid misrecognition of colors, an application can do the following:

1. When the application loads, it should use the **GetSystemColors** function to retrieve the system colors from the system palette and compare these colors against the system colors used in the DIB palettes.
2. If the colors do not match, the application should copy the current system colors (retrieved from the system palette) over the DIB system colors.

23.3 Using a Device-Independent Bitmap Driver

Many MS-DOS applications manipulate screen memory directly. To maintain the device independence of Windows, it is not possible to allow an application to access screen memory directly. However, an application can use the DIB driver (DIB.DRV) to directly manipulate an image in memory.

23.3.1 Creating a Driver Display Context

An application can load the DIB driver by passing the DIB driver name and a **BITMAPINFO** structure containing the DIB bits to the **CreateDC** function. For example, the following example creates a DIB display context that represents the packed DIB described by the **BITMAPINFO** structure *bi*:

```
hdc = CreateDC("DIB", NULL, NULL, &bi);
```

An application must observe the following rules when working with a device context created in this manner:

- If the last parameter of **CreateDC** is NULL, the display context is associated with a 0-by-0 8-bit DIB. Any attempt to draw with it will fail.

- The **BITMAPINFO** structure must remain locked for the life of the device context.
- The DIB driver supports 1-bit, 4-bit, or 8-bit DIB bitmaps. The run-length encoding (RLE) format is not supported.
- The DIB driver supports only Windows version 3.0 or later DIB headers.
- Multiple DIB-driver display contexts can be active.
- DIBs reside in the memory-based image buffer in the CF_DIB (packed-DIB) format.
- The DIB driver expects the **RGBQUAD** structure for color matching; it does not use palette indices. (If an application uses an RGB value for drawing, the DIB driver uses the closest match found in the color table of the DIB.)

The following example uses the DIB driver to draw a circle in a DIB copied from the clipboard:

```
if (IsClipboardFormatAvailable(CF_DIB) && OpenClipboard()) {
    HANDLE hdib;
    HDC    hdc;

    /* Get the DIB from the clipboard.                 */

    hdib = GetClipboardData(CF_DIB);

    /* Create a DIB driver hdc on the DIB surface. */

    hdc = CreateDC("DIB", NULL, NULL,
        (LPBITMAPINFO) GlobalLock(hdib));

    /* Draw a circle in the DIB.                       */

    Ellipse(hdc, 0, 0, 100, 100);

    /* Delete the DIB driver HDC now that you are done with it. */

    DeleteDC(hdc);

    /* Unlock the DIB.                                 */

    GlobalUnlock(hdib);

    /* Release the clipboard.                          */

    CloseClipboard();
}
```

23.3.2 Moving Bitmaps to and from the Display

The DIB driver is a separate driver and is not associated with the display driver. Because of this, an application cannot use the **BitBlt** function to move bitmaps between a DIB-driver device context and a screen device context. An application can use the **GetDIBits** function to copy from the screen device context to a DIB device context. To copy a DIB device context to the screen device context, an application can use the **StretchDIBits** function.

An application can maximize the speed of **StretchDIBits** by using one of the following methods:

- One-to-one mapping for the palette
- DIB_PAL_COLORS, an option that prevents color matching by the graphics device interface (GDI)

23.3.3 Modifying Bitmaps

DIBs offer many advantages over device-dependent bitmaps. Unlike device-dependent bitmaps, however, DIBs cannot be selected into a video device context. Before the DIB driver was available, this meant that applications could not take advantage of the extensive graphics device interface (GDI) functions to modify DIBs directly. To use GDI routines to draw in or otherwise modify a DIB, an application would follow a procedure such as this:

1. Create a memory device context.
2. Use the **CreateDIBitmap** function to convert the DIB to device-dependent format.
3. Select the device-dependent bitmap into the memory device context.
4. Call GDI routines to modify the device-dependent bitmap.
5. Use the **GetDIBits** function to convert the device-dependent bitmap to DIB format.

This method works well if you only use GDI routines to modify the bitmap. If you want to speed up certain operations by writing replacement functions that directly modify the DIB bits, however, the procedure can become complicated. The direct-manipulation routines work on the DIB, but the GDI routines work on the device-dependent bitmap.

Direct manipulation can be considerably faster than using equivalent GDI routines; in one sample application, a direct-manipulation function (drawing a triangle) ran eight times faster than the equivalent GDI operation. Also, direct-manipulation routines for other products may be reusable.

The DIB driver makes it possible for you to mix GDI calls with direct-manipulation routines, so you can combine the advantages of both methods.

23.3.4 Creating a Driver Device Context

The DIB driver makes it possible for you to create a DIB device context. To create the DIB device context, call the **CreateDC** function, supplying a pointer to a **BITMAPINFO** structure:

```
hdc = CreateDC("DIB", NULL, NULL, lpbi);
```

You can use the device-context handle returned by the **CreateDC** function with most GDI functions to modify the bitmap. Concurrently, you can call your own direct-manipulation functions to modify the actual bitmap bits. Any changes made directly to the bitmap bits are reflected in the DIB-driver device context. When you finish modifying the bitmap, you can use the **StretchDIBits** function to transfer the DIB to the video device context.

The DIB driver can handle 1-bit, 4-bit, or 8-bit DIBs. You can create multiple DIB driver contexts. Note the following limitations:

1. The **BITMAPINFO** structure must be locked for the life of the device context.
2. The DIB driver handles only the Windows **BITMAPINFOHEADER** format.
3. The RLE format is not supported.
4. The DIB must use the DIB_RGB_COLORS format. The DIB driver does not support the DIB_PAL_COLORS (palette indexes) format.

You can distribute the DIB driver with applications that run under Windows.

Self-Loading Windows Applications

Chapter 24

This chapter describes the contents of a unique segment that is found only in self-loading applications for the Microsoft Windows operating system. This segment contains six functions: three that the application developer supplies and three that the Windows kernel supplies. The segment also contains a table of pointers to these functions and loader code.

This chapter contains references to the Windows (new-style) header and the data tables in a Windows executable file. For a complete description of an executable file before it is altered by the loader and loaded into memory, see the *Microsoft Windows Programmer's Reference, Volume 4.*

24.1 Loader Functions

The Windows kernel provides a loader function that places applications into memory and passes execution to a specified entry point. Some Windows applications, however, must bypass this kernel function and load themselves in order to be executed correctly. For example, a compiler for Windows might contain two floating-point modules: one requiring a math coprocessor and one emulating the coprocessor. The standard loader function in the Windows kernel does not provide a method of specifying that code in one module should be loaded in place of code in another; this means that the compiler needs to load the appropriate code itself in order to run efficiently and correctly. Likewise, the code for a Windows application might be compressed with a special compression algorithm in order to fit on a certain number of disks, but the standard loader function does not provide a method for dealing with a compressed file format. The application, therefore, must load itself in order to be executed correctly.

To indicate that a Windows application is self-loading, the 16-bit flag value in the executable file's Windows header must contain the value 0x0800 (that is, bit 11 must be set). Otherwise, Windows ignores the private loader code and installs the application by using the standard loader functions in the Windows kernel.

24.2 Loader Data Table

In addition to the loader functions, the first segment of a self-loading Windows application contains a loader data table with far pointers to each of the loader functions. The format of this table follows:

Location	Description
0x00	Specifies the version number (this value must be 0xA0).
0x02	Reserved.
0x04	Points to a startup procedure, which the application developer provides.

Location	Description
0x08	Points to a reloading procedure, which the application developer provides.
0x0C	Reserved.
0x10	Points to a memory-allocation procedure, which the kernel provides.
0x14	Points to an entry-number procedure, which the kernel provides.
0x18	Points to an exit procedure, which the application developer provides.
0x1C	Reserved.
0x1E	Reserved.
0x20	Reserved.
0x22	Reserved.
0x24	Points to a set-owner procedure, which the kernel provides.

All of the pointers in this table must point to locations within the first segment. There can be no fixups outside this segment.

After the segment table for an executable file is loaded into memory, each entry contains an additional 16-bit value. This value is a segment selector (or handle) that the loader created.

24.3 Loader Code

The first segment of a self-loading Windows application contains loader code for the six required loader functions. The code loads and reloads segments and resets hardware.

24.3.1 Loading Segments

The kernel calls the **BootApp** function supplied by the application developer, instead of loading the application in the normal manner, if the 16-bit value in the information block for the Windows header contains the value 0x0800 (that is, bit 11 is set). The **BootApp** function allocates memory for all segments by calling the kernel-supplied **MyAlloc** function. If the segment is identified as a **PRELOAD** or **FIXED** type, **BootApp** also calls the **LoadAppSeg** function (another function supplied by the application developer). The **BootApp** function also calls **SetOwner**, a kernel-supplied function, to associate the correct information block with each segment handle.

The first segment that the **BootApp** function should allocate is the application's automatic data segment. This data segment contains the application's stack. The automatic data segment must be allocated before the **BootApp** function calls the Windows **PatchCodeHandle** function. For more information about the **Patch-CodeHandle** function, see the *Microsoft Windows Programmer's Reference, Volume 2.*

24.3.2 Reloading Segments

In addition to loading segments, the **LoadAppSeg** function reloads segments that the Windows kernel has discarded. Because the **LoadAppSeg** function is responsible for reloading segments, it must update bits 1 and 2 of the 16-bit flag value in the segment table. (Only self-loading applications should alter the Windows header or the data tables that follow it.) Bit 1 specifies whether memory is allocated for the segment, and bit 2 specifies whether the segment is currently loaded. For a complete description of the segment table, see the *Microsoft Windows Programmer's Reference, Volume 4.*

If the loader allocates memory for a segment but the segment is not loaded (that is, bit 1 is set and bit 2 is not), the **LoadAppSeg** function should call the Windows **GlobalHandle** function to determine whether memory is allocated for the segment. If memory is not allocated, the **LoadAppSeg** function should call the Windows **GlobalReAlloc** function to reallocate memory for the segment.

Once memory is allocated, the **LoadAppSeg** function should read the segment from the executable file and call the **PatchCodeHandle** function to correct each function prolog that occurs in the segment. Once the function prologs are altered, the **LoadAppSeg** function should resolve any far pointers that occur in the segment. If the pointer is specified by an ordinal value, the **LoadAppSeg** function should call the kernel-supplied **EntryAddrProc** function to resolve the address.

24.3.3 Resetting Hardware

When closing a self-loading application, the kernel calls the **ExitProc** function, supplied by the application developer, to reset any hardware that a dynamic-link library may have accessed. However, the **ExitProc** function does not need to free memory or close files.

24.4 Function Reference

This section provides information about the functions supplied by the application developer and by the kernel for self-loading Windows applications.

BootApp

3.1

```
void BootApp(hBlock, hFile)
HANDLE hBlock;    /* handle of information block    */
HANDLE hFile;     /* handle of executable file      */
```

The **BootApp** function loads the given application.

Parameters

hBlock
Identifies the selector for the segment that contains the information block in the Windows (new-style) header.

hFile
Identifies the executable file that contains the application. The *hFile* parameter must be a valid MS-DOS file handle.

Return Value

This function does not return a value.

Comments

The information block in the Windows header that is identified by the *hBlock* parameter specifies the linker version number, the length of various tables of data, offsets to those tables, heap and stack sizes, and so on. For a description of the Windows header, see the *Microsoft Windows Programmer's Reference, Volume 4.*

The **BootApp** function is one of three functions required for self-loading Windows applications. The application developer must provide the code for this function and store a pointer to the function at offset 0x0004 in the application's loader code and data table.

The Windows kernel calls this function after loading the application's executable header and data tables.

EntryAddrProc

3.1

DWORD EntryAddrProc(*hBlock*, *wEntryNo***)**
HANDLE *hBlock*; /* selector for information block */
WORD *wEntryNo*; /* entry-table procedure index */

The **EntryAddrProc** function retrieves an address for the specified procedure.

Parameters

hBlock
Specifies the selector for the segment that contains the information block in the Windows (new-style) header.

wEntryNo
Specifies the index to the entry in an entry table that identifies the procedure for which the function should return an address.

Return Value

The return value is the address of the specified procedure if the function is successful. Otherwise, the return value is zero.

Comments

The *wEntryNo* parameter is also known as the procedure's ordinal number.

The **EntryAddrProc** function is one of three functions supplied by the Windows kernel. The kernel loads a pointer to this function at offset 0x0014 in the loader's code and data table. The kernel loads the pointer before calling the private startup procedure (the **BootApp** function).

EntryAddrProc is called from the **LoadAppSeg** function, which the application developer must supply.

ExitProc

3.1

void ExitProc(*hBlock***)**
HANDLE *hBlock*; /* selector of information block */

The **ExitProc** function closes a self-loading application.

Parameters

hBlock
Specifies the selector for the segment that contains the information block in the Windows (new-style) header.

Return Value

This function does not return a value.

Comments

The Windows header information block identified by the *hBlock* parameter specifies the linker version number, the length of various tables of data, offsets to those tables, heap and stack sizes, and so on. For a description of the Windows header, see the *Microsoft Windows Programmer's Reference, Volume 4.*

The **ExitProc** function is one of three functions required for self-loading Windows applications. The application developer must provide the code for this function and store a pointer to it at offset 0x0018 in the application's loader code and data table.

ExitProc does not need to free memory owned by the application, nor is it necessary for the function to close any open files.

LoadAppSeg

3.1

```
WORD LoadAppSeg(hBlock, hFile, wSegID)
HANDLE hBlock;    /* handle of module information block     */
HANDLE hFile;     /* handle of executable file              */
WORD  wSegID;     /* segment identifier                     */
```

The **LoadAppSeg** function loads a segment for the first time or reloads a discarded segment. The segment is identified by the *wSegID* parameter and belongs to the given application.

Parameters

hBlock
Specifies the segment selector for the segment containing the module information block.

hFile
Identifies the executable file that contains the application. This parameter is an MS-DOS file handle. (This handle is –1 if the file is not open.)

wSegID
Identifies the segment that the function should reload.

Return Value

The return value is a selector for the segment if the function is successful. Otherwise, it is zero.

Comments

The information block in the Windows (new-style) header identified by the *hBlock* parameter specifies the linker version number, the length of various tables of data, offsets to those tables, heap and stack sizes, and so on. For a description of the Windows header, see the *Microsoft Windows Programmer's Reference, Volume 4.*

The third parameter, *wSegID*, is determined by the linker at link time.

The **LoadAppSeg** function is one of three functions required for self-loading Windows applications. The application developer must provide the code for this function and store a pointer to it at offset 0x0008 in the application's loader code and data table.

MyAlloc

3.1

DWORD MyAlloc(*wFlags*, *wSize*, *wElem***)**
WORD *wFlags*; /* segment flags */
WORD *wSize*; /* size of element */
WORD *wElem*; /* number of elements in segment */

The **MyAlloc** function allocates memory for a segment in a self-loading application.

Parameters

wFlags
Specifies the segment flags.

wSize
Specifies the element size, in bytes.

wElem
Specifies the number of elements in the segment.

Return Value

The low-order word of the return value contains a segment handle if the function is successful; the high-order word contains a selector if the function is successful. (However, if the function allocates only a handle for the segment, the low-order word contains zero and the high-order word contains the handle.) Otherwise, the return value is zero for both high-order and low-order words.

Comments

The flags specified by the *wFlags* parameter are the values that precede the segment table appearing immediately after the information block in the Windows (new-style) header. The kernel translates *wFlags* into the proper values before calling the **GlobalAlloc** function.

The segment size, in bytes, is obtained by shifting the value specified in the *wSize* parameter left by the number of bits specified by the *wElem* parameter.

The **MyAlloc** function is one of three functions supplied by the Windows kernel. The kernel loads a pointer to this function at offset 0x0014 in the loader's code and data table. The kernel loads the pointer before calling the private startup procedure (the **BootApp** function).

See Also

BootApp

SetOwner

3.1

void SetOwner(*hSel*, *hOwner*)
WORD *hSel*; /* selector of segment */
HANDLE *hOwner*; /* handle of information block */

The **SetOwner** function associates the given segment with an executable file or application.

Parameters

hSel
Specifies a selector or handle identifying the segment to be associated with the executable file or application.

hOwner
Identifies the information bock in the Windows (new-style) executable-file header for the application that contains the segment.

Return Value

This function does not return a value.

Comments

The Windows header information block identified by the *hOwner* parameter specifies the linker version number, the length of various tables of data, offsets to these tables, heap and stack sizes, and so on. For a description of the Windows header, see the *Microsoft Windows Programmer's Reference, Volume 4*.

The **SetOwner** function is one of three functions required for self-loading Windows applications. The application developer must provide the code for this function and store a pointer to it at offset 0x0004 in the application's loader code and data table.

After the kernel allocates memory for a segment by using the **MyAlloc** function, it calls **SetOwner**.

Installable Drivers

Chapter 25

This chapter describes installable drivers and the installable-driver interface for the Microsoft Windows operating system. Topics discussed in this chapter include: the common entry point for installable drivers, messages used by the common entry point, actions that an installable driver should take in response to these messages, and functions available for the installable driver interface.

25.1 About Installable Drivers

An installable driver is a Windows dynamic-link library (DLL) that a Windows application (or another Windows DLL) can open, enable, query, disable, and close. An application can perform these operations by calling the following functions:

Function	Description
CloseDriver	Closes an installable driver.
GetDriverInfo	Retrieves installable-driver data.
GetDriverModuleHandle	Retrieves an installable driver's module handle.
GetNextDriver	Enumerates installed drivers.
OpenDriver	Opens an installable driver.
SendDriverMessage	Sends a message to an installable driver.

When an application calls the **OpenDriver**, **SendDriverMessage**, or **CloseDriver** function, Windows processes the call and issues one or more of the following driver messages:

Message	Description
DRV_CLOSE	Notifies an installable driver that Windows will decrement the use count for the driver and send a DRV_FREE message if the use count reaches zero.
DRV_CONFIGURE	Notifies an installable driver that it should display a custom-configuration dialog box. (This message should only be sent if the driver returns a nonzero value when the DRV_QUERYCONFIGURE message is processed.)
DRV_DISABLE	Notifies an installable driver that the memory that it has allocated is about to be freed.
DRV_ENABLE	Notifies an installable driver that it has been loaded or reloaded or that Windows has been enabled.
DRV_FREE	Notifies an installable driver that it will be discarded.
DRV_INSTALL	Notifies an installable driver that it has been successfully installed.
DRV_LOAD	Notifies an installable driver that it has been successfully loaded.

Message	Description
DRV_OPEN	Notifies an installable driver that it is about to be opened.
DRV_POWER	Notifies an installable driver that the power source for the device is about to be turned off or on.
DRV_QUERYCONFIGURE	Queries an installable driver about whether it supports the DRV_CONFIGURE message and can display a private configuration dialog box.
DRV_REMOVE	Notifies an installable driver that it is about to be removed from the system.

These messages, which are defined in the Windows header file (WINDOWS.H), are processed by the main routine in an installable driver. This routine is called the **DriverProc** function.

Some of the preceding messages should be sent by Windows only when one of the installable driver functions is called by an application. The circumstances under which these messages are sent are described in the following list:

Message	Description
DRV_CLOSE	Issued by Windows when an application calls the **CloseDriver** function.
DRV_DISABLE	Issued prior to exiting Windows and returning to MS-DOS or when the driver is freed.
DRV_ENABLE	Issued when returning to Windows from MS-DOS or the first time the installable driver is loaded.
DRV_FREE	Issued by Windows after an application calls the **CloseDriver** function and the use count is decremented to zero.
DRV_LOAD	Issued by Windows after the first **OpenDriver** call is made for a particular installable driver.

The remaining messages can be sent by an application to an installable driver by calling the **SendDriverMessage** function.

25.2 Creating an Installable Driver

An installable driver is a Windows dynamic-link library (DLL) that supports a special entry point, the **DriverProc** function. This function processes the driver messages described in the previous section. This function may also process private driver messages. These messages can be assigned values ranging from DRV_RESERVED to DRV_USER (two constants that appear in WINDOWS.H).

The following example shows the basic structure of the **DriverProc** function:

```
LRESULT CALLBACK* DriverProc (DWORD  dwDriverIdentifier,
                              HDRVR  hDriver,
                              UINT   wMessage,
                              LPARAM lParam1,
                              LPARAM lParam2)
{
DWORD dwRes = 0L;

switch (wMessage)
    {

    case DRV_LOAD:

        /* Sent when the driver is loaded. This is always   */
        /* the first message received by a driver.          */

        dwRes = 1L;     /* Return 0L to fail.               */
        break;

    case DRV_FREE:

        /* Sent when the driver is about to be discarded.   */
        /* This is the last message a driver receives       */
        /* before it is freed.                              */

        dwRes = 1L;     /* Return value is ignored.         */
        break;

    case DRV_OPEN:

        /* Sent when the driver is opened.                  */

        dwRes = 1L;     /* Return 0L to fail.               */
                        /* This value is subsequently used  */
                        /* for dwDriverIdentifier.          */

        break;

    case DRV_CLOSE:

        /* Sent when the driver is closed. Drivers are      */
        /* unloaded when the open count reaches zero.       */

        dwRes = 1L;     /* Return 0L to fail.               */
        break;
```

```
case DRV_ENABLE:

    /* Sent when the driver is loaded or reloaded and   */
    /* when Windows is enabled. Hook or rehook          */
    /* interrupts and initialize hardware. Expect the   */
    /* driver to be in memory only between the enable   */
    /* and disable messages.                            */

    dwRes = 1L;     /* Return value is ignored.         */
    break;

case DRV_DISABLE:

    /* Sent before the driver is freed or when Windows  */
    /* is disabled. Unhook interrupts and place         */
    /* peripherals in an inactive state.                */

    dwRes = 1L;     /* Return value is ignored.         */
    break;

case DRV_INSTALL:

    /* Sent when the driver is installed.               */

    dwRes = DRV_OK; /* Can also return DRV_CANCEL       */
                    /* and DRV_RESTART.                 */
    break;

case DRV_REMOVE:

    /* Sent when the driver is removed.                 */

    dwRes = 1L;     /* Return value is ignored.         */
    break;

case DRV_QUERYCONFIGURE:

    /* Sent to determine if the driver can be           */
    /* configured.                                      */

    dwRes = 0L;     /* Zero indicates configuration     */
                    /* NOT supported.                   */
    break;

case DRV_CONFIGURE:

    /* Sent to display the custom-configuration         */
    /* dialog box for the driver.                       */

    dwRes = DRV_OK; /* Can also return DRV_CANCEL       */
                    /* and DRV_RESTART.                 */
    break;
```

```
        default:

            /* Process any messages not explicitly trapped.     */

            return DefDriverProc (dwDriverIdentifier, hDriver,
                                  wMessage, lParam1, lParam2);

        }
    return dwRes;
    }
```

25.2.1 Opening an Installable Driver

An application opens an installable driver by calling the **OpenDriver** function. When an application calls this function, Windows adds the driver name to an internal list of installed drivers. (When the application calls the **CloseDriver** function, Windows deletes the corresponding driver name from this list.)

When an application calls the **OpenDriver** function to open the first instance of a driver, Windows issues the DRV_LOAD, DRV_ENABLE, and DRV_OPEN messages, in that order. (Subsequent calls to **OpenDriver** cause only DRV_OPEN to be sent.) When the driver processes the DRV_LOAD message, it reads the configuration settings (if any exist) from the corresponding entry in the SYSTEM.INI file and configures the driver and any associated hardware. In addition to configuring the driver and associated hardware, the driver also allocates required memory.

After processing the DRV_LOAD message, the driver returns a nonzero value if it loads successfully. If it returns zero, Windows immediately unloads the driver (without issuing a DRV_FREE message).

When the driver processes the DRV_ENABLE message, it hooks or chains required interrupts and prepares associated peripherals.

When the driver processes the DRV_OPEN message, it allocates memory required by a single instance of the driver.

25.2.2 Closing an Installable Driver

An application closes an installable driver by calling the **CloseDriver** function. When the application calls this function, Windows deletes the corresponding driver name from an internal list.

When an application calls the **CloseDriver** function to close the last instance of a driver, Windows issues the DRV_CLOSE, DRV_DISABLE, and DRV_FREE messages, in that order. (When the application is not closing the last instance of the driver, only DRV_CLOSE is sent.) When the driver processes the

DRV_CLOSE message, it frees any resources that were allocated when the driver was opened and returns a nonzero value. If the driver returns a value of zero, closing fails.

When the driver processes the DRV_DISABLE message, it places any associated peripherals in an inactive state and unhooks all interrupts.

When the driver processes the DRV_FREE message, it frees any resources that are still allocated.

25.2.3 Configuring an Installable Driver

Many installable drivers support a private configuration dialog box that lets the user configure the driver and associated hardware. To determine whether a driver supports such a dialog box, an application calls the **SendDriverMessage** function and issues the DRV_QUERYCONFIGURE message. If the driver is configurable, this function returns a nonzero value. If it is not configurable, this function returns zero. If the **SendDriverMessage** function returns a nonzero value, the application displays the configuration dialog box by calling the **SendDriverMessage** function a second time and sending the DRV_CONFIGURE message.

If the driver supports a private configuration dialog box, it should display the dialog box and process user input when it receives the DRV_CONFIGURE message. Typically, any configuration data specified by the user is maintained in the [drivers] section of the Windows SYSTEM.INI file.

25.2.4 Enumerating Instances of an Installable Driver

An application can retrieve a handle identifying either the first instance of an installable driver or each instance of the driver by calling the **GetNextDriver** function.

25.3 Updating the SYSTEM.INI File

Upon installation, the [drivers] section of the SYSTEM.INI file contains an entry for each installable driver. This entry has the following form:

entry=driver_filename optional_information

An application can open a driver by using its filename or its entry. If a fully qualified path is not specified with the filename, the driver file must exist on the standard Windows search path. The driver interface searches for the driver as follows:

- If an application specifies a section name, that section of SYSTEM.INI is searched instead of the [drivers] section.
- If an application specifies an entry in the search section, the driver with a filename corresponding to the entry is opened.
- If the string specified by the application does not match an entry in the search section, the system assumes the string is a driver filename.

The optional information (*optional_information*) following the driver name (*driver_filename*) lists information a driver needs after installation. A driver maintains configuration information here if the information is limited or if it needs to be associated with the entry. For example, two prototype drivers could be installed in the system. The first driver could be associated with serial port one, and the second driver could be associated with serial port two. The [drivers] section of the SYSTEM.INI might show this association in the following way:

```
[drivers]
prototype1=proto.drv com1
prototype2=proto.drv com2
```

If your driver uses more extensive configuration information, it can create a section in the SYSTEM.INI file reserved for its parameters. For example, the installable driver PROTO.DRV might create the following [proto.drv] section:

```
[proto.drv]
port=230
int=3
```

When reserving a section for your driver, use the filename of your driver to identify the section. A driver usually configures and maintains this section of information when it displays the configuration dialog box used for the DRV_CONFIGURE message.

If you want your installable driver loaded when Windows starts, place its filename or an alias from the [drivers] section of the SYSTEM.INI file on the drivers= line of the [boot] section found in the SYSTEM.INI file. Windows loads these drivers at startup and sends DRV_LOAD and DRV_ENABLE messages to them but does not open them. This makes it possible for you to install drivers that remain resident while Windows is enabled.

25.4 Contents of the OEMSETUP.INF Files

The OEMSETUP.INF file uses the same format as the Windows 3.0 SETUP.INF file with the exception of a new [Installable.Drivers] section. This section identifies the names and characteristics of each driver on the disk. Each driver entry has the following form:

entry = disk:filename, type(s), description, VxD(s), default_params

Note that the elements that compose a driver entry are separated by commas. Comments are delimited by semicolons; all characters following a semicolon are considered part of the comment string.

Following are the elements that compose a driver entry:

Element	Description
entry	Identifies the driver. This string must be unique.
disk	Specifies the disk number for the disk that contains the driver. This entry corresponds to an entry in the [disks] section of SETUP.INF.
filename	Specifies the name of the file that contains the driver.
type(s)	Specifies the driver type.
description	Describes the driver. This string appears in the dialog box displayed by the Drivers Control Panel application.
VxD(s)	Identifies any VxDs required by the driver. (For a description of the manner in which multiple VxD names are parsed, see the *Microsoft Windows Virtual Device Adaptation Guide*.)
default_params	Specifies default parameters for the driver. Additional options are appended to the driver entry in the [drivers] section of SYSTEM.INI.

If you create an OEMSETUP.INF file to distribute with your driver, it must include the [disks] and [Installable.Drivers] sections. For example, the following entries could be used in an OEMSETUP.INF file for a prototype installable driver:

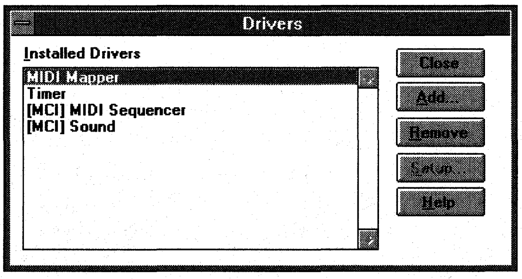
```
[disks]
; Numeric mappings for disk titles

1 = ., "Sample Distribution Disk 1"

[Installable.Drivers]
; The installable drivers section is unique to the drivers application.
; It is parsed with comma-separated fields.

prototype=1:proto.drv,"ampl,freq","Sample scope driver","1:VXDA.386"
```

The Drivers Control Panel application may need to copy files that support your driver. If any of these files are not VxDs, include a section in the SYSTEM.INI file listing them. Use the entry (that is, prototype) as the name of this new section. For example, if the prototype driver has an additional file called POWERSRC.DLL, include the following section:

```
[prototype]
; Keyname sections can be created for dependent files.  All
; dependent files will be copied directly to the system directory.

1:POWERSRC.DLL
```

25.5 Drivers Control Panel Application

The Drivers Control Panel application installs, configures, and removes drivers. When started, the Drivers Control Panel application displays the following dialog box.



The Installed Drivers list box displays the description strings of the installed drivers. The installed drivers are determined by examining the [drivers] and [mci] sections of the SYSTEM.INI file. The description strings are cached in the [drivers.description] section of the CONTROL.INI file to reduce delays in finding and loading them. If a description string does not match an installed driver, the application searches the MMSETUP.INF file and then the header of the driver file to obtain the description string. A scroll bar appears in the list box if there are more drivers than can be displayed.

The following buttons are found in the Control Panel dialog box:

Button	Result when chosen
OK	Exits the dialog box and makes any changes permanent.
Cancel	Exits the dialog box. The application ignores any requests to install or remove drivers made during the session. Any configuration changes made during the session are retained because they are done by the driver.

Button	Result when chosen
Remove	Removes the information about the selected driver from the SYSTEM.INI file. When removing drivers, the Control Panel application sends the DRV_REMOVE message to the driver if there is only one entry in the SYSTEM.INI file for it.
Setup	Applies only to configurable drivers. When the user selects a driver in the list box, the application opens the driver and sends it the DRV_QUERYCONFIGURE message. If a driver responds that it can be configured—that is, it supports a configuration dialog box to set such parameters as the COM port, the interrupt number, or input and output (I/O) port address—then the application enables the Setup button. If the user chooses the Setup button, the application sends a DRV_CONFIGURE message to the driver.
Add Drivers	Installs a new driver.
Default	Redisplays the list of files from the MMSETUP.INF file. Note that the Default button is active when the OEM drivers are displayed.

25.5.1 Installing a Driver

When the user selects a driver from the Installed Drivers list box, the Add Driver dialog box closes. The new driver becomes selected in the list box when the user chooses the OK button. The Drivers Control Panel application sends the DRV_INSTALL message to the driver if there is only one entry in the SYSTEM.INI file for it. (A driver receives the DRV_INSTALL message for its initial installation.) The Drivers Control Panel application can install up to four wave devices, four musical instrument digital interface (MIDI) devices, and ten media control interface (MCI) devices of the same type.

If the selected driver is not an installable driver, the Driver Control Panel applications displays a "Cannot Install" message. If the user chooses the Cancel button, the dialog box closes with no changes made.

25.5.2 Using Drivers with the Drivers Control Panel Application

During installation, the Drivers Control Panel application opens the driver and obtains the description line, originally defined in the module-definition (.DEF) file, from the driver header. The application uses the description line to construct the settings for the [drivers] section. The description line in the .DEF file should have the following form:

DESCRIPTION *type(s)*:*text*

Following are the parameters in the description line:

Parameter	Meaning
type(s)	Type of driver used for the entry in the SYSTEM.INI file. Multiple entries are separated by commas.
text	Text that describes the driver. This will be displayed in the Drivers Control Panel application.

For example, the header file for an oscilloscope driver (OSCI.DRV) can use the following description line:

```
DESCRIPTION 'FREQ,AMPL:Oscilloscope frequency and amplitude drivers.'
```

Based on this definition, if both drivers are installed (that is, if the Drivers Control Panel application displays a selection for both FREQ and AMPL), the Drivers Control Panel application creates the following settings in the SYSTEM.INI file:

```
[drivers]
FREQ = osci.drv
AMPL = osci.drv
```

If you want your driver added to a named section of the SYSTEM.INI file, you can add the section name to the type of driver. For example, the following description line specifies that a voltmeter driver be added to the [RCC] section:

```
DESCRIPTION 'VOLTMETER[RCC]:RCC voltmeter driver.'
```

25.6 Creating a Custom Configuration Application

The Drivers Control Panel application provides a convenient interface for installing drivers. You should use this interface for configuring features that are hardware- or driver-dependent.

If your driver configures system features—those features that are hardware- and device-independent—you should create a custom Control Panel application.

Appendix

Module and Library Names

Appendix

This appendix lists the module and import libraries associated with each Microsoft Windows function.

Function	Module	Import library
AbortDoc	GDI	LIBW.LIB
AccessResource	KERNEL	LIBW.LIB
AddAtom	KERNEL	LIBW.LIB
AddFontResource	GDI	LIBW.LIB
AdjustWindowRect	USER	LIBW.LIB
AdjustWindowRectEx	USER	LIBW.LIB
AllocDiskSpace	STRESS	STRESS.LIB
AllocDStoCSAlias	KERNEL	LIBW.LIB
AllocFileHandles	STRESS	STRESS.LIB
AllocGDIMem	STRESS	STRESS.LIB
AllocMem	STRESS	STRESS.LIB
AllocResource	KERNEL	LIBW.LIB
AllocSelector	KERNEL	LIBW.LIB
AllocUserMem	STRESS	STRESS.LIB
AnimatePalette	GDI	LIBW.LIB
AnsiLower	USER	LIBW.LIB
AnsiLowerBuff	USER	LIBW.LIB
AnsiNext	USER	LIBW.LIB
AnsiPrev	USER	LIBW.LIB
AnsiToOem	KEYBOARD	LIBW.LIB
AnsiToOemBuff	KEYBOARD	LIBW.LIB
AnsiUpper	USER	LIBW.LIB
AnsiUpperBuff	USER	LIBW.LIB
AnyPopup	USER	LIBW.LIB
AppendMenu	USER	LIBW.LIB
Arc	GDI	LIBW.LIB
ArrangeIconicWindows	USER	LIBW.LIB
BeginDeferWindowPos	USER	LIBW.LIB
BeginPaint	USER	LIBW.LIB
BitBlt	GDI	LIBW.LIB
BringWindowToTop	USER	LIBW.LIB
BuildCommDCB	USER	LIBW.LIB
CallMsgFilter	USER	LIBW.LIB
CallNextHookEx	USER	LIBW.LIB
CallWindowProc	USER	LIBW.LIB

Function	Module	Import library
Catch	KERNEL	LIBW.LIB
ChangeClipboardChain	USER	LIBW.LIB
ChangeMenu	USER	LIBW.LIB
CheckDlgButton	USER	LIBW.LIB
CheckMenuItem	USER	LIBW.LIB
CheckRadioButton	USER	LIBW.LIB
ChildWindowFromPoint	USER	LIBW.LIB
ChooseColor	COMMDLG	COMMDLG.LIB
ChooseFont	COMMDLG	COMMDLG.LIB
Chord	GDI	LIBW.LIB
ClassFirst	TOOLHELP	TOOLHELP.LIB
ClassNext	TOOLHELP	TOOLHELP.LIB
ClearCommBreak	USER	LIBW.LIB
ClientToScreen	USER	LIBW.LIB
ClipCursor	USER	LIBW.LIB
CloseClipboard	USER	LIBW.LIB
CloseComm	USER	LIBW.LIB
CloseDriver	USER	LIBW.LIB
CloseMetaFile	GDI	LIBW.LIB
CloseWindow	USER	LIBW.LIB
CombineRgn	GDI	LIBW.LIB
CommDlgExtendedError	COMMDLG	COMMDLG.LIB
CopyCursor	USER	LIBW.LIB
CopyIcon	USER	LIBW.LIB
CopyLZFile	LZEXPAND	LZEXPAND.LIB
CopyMetaFile	GDI	LIBW.LIB
CopyRect	USER	LIBW.LIB
CountClipboardFormats	USER	LIBW.LIB
CreateBitmap	GDI	LIBW.LIB
CreateBitmapIndirect	GDI	LIBW.LIB
CreateBrushIndirect	GDI	LIBW.LIB
CreateCaret	USER	LIBW.LIB
CreateCompatibleBitmap	GDI	LIBW.LIB
CreateCompatibleDC	GDI	LIBW.LIB
CreateCursor	USER	LIBW.LIB
CreateDC	GDI	LIBW.LIB
CreateDialog	USER	LIBW.LIB
CreateDialogIndirect	USER	LIBW.LIB

Function	Module	Import library
CreateDialogIndirectParam	USER	LIBW.LIB
CreateDialogParam	USER	LIBW.LIB
CreateDIBitmap	GDI	LIBW.LIB
CreateDIBPatternBrush	GDI	LIBW.LIB
CreateDiscardableBitmap	GDI	LIBW.LIB
CreateEllipticRgn	GDI	LIBW.LIB
CreateEllipticRgnIndirect	GDI	LIBW.LIB
CreateFont	GDI	LIBW.LIB
CreateFontIndirect	GDI	LIBW.LIB
CreateHatchBrush	GDI	LIBW.LIB
CreateIC	GDI	LIBW.LIB
CreateIcon	USER	LIBW.LIB
CreateMenu	USER	LIBW.LIB
CreateMetaFile	GDI	LIBW.LIB
CreatePalette	GDI	LIBW.LIB
CreatePatternBrush	GDI	LIBW.LIB
CreatePen	GDI	LIBW.LIB
CreatePenIndirect	GDI	LIBW.LIB
CreatePolygonRgn	GDI	LIBW.LIB
CreatePolyPolygonRgn	GDI	LIBW.LIB
CreatePopupMenu	USER	LIBW.LIB
CreateRectRgn	GDI	LIBW.LIB
CreateRectRgnIndirect	GDI	LIBW.LIB
CreateRoundRectRgn	GDI	LIBW.LIB
CreateScalableFontResource	GDI	LIBW.LIB
CreateSolidBrush	GDI	LIBW.LIB
CreateWindow	USER	LIBW.LIB
CreateWindowEx	USER	LIBW.LIB
DdeAbandonTransaction	DDEML	DDEML.LIB
DdeAccessData	DDEML	DDEML.LIB
DdeAddData	DDEML	DDEML.LIB
DdeClientTransaction	DDEML	DDEML.LIB
DdeCmpStringHandles	DDEML	DDEML.LIB
DdeConnect	DDEML	DDEML.LIB
DdeConnectList	DDEML	DDEML.LIB
DdeCreateDataHandle	DDEML	DDEML.LIB
DdeCreateStringHandle	DDEML	DDEML.LIB
DdeDisconnect	DDEML	DDEML.LIB

Function	Module	Import library
DdeDisconnectList	DDEML	DDEML.LIB
DdeEnableCallback	DDEML	DDEML.LIB
DdeFreeDataHandle	DDEML	DDEML.LIB
DdeFreeStringHandle	DDEML	DDEML.LIB
DdeGetData	DDEML	DDEML.LIB
DdeGetLastError	DDEML	DDEML.LIB
DdeInitialize	DDEML	DDEML.LIB
DdeKeepStringHandle	DDEML	DDEML.LIB
DdeNameService	DDEML	DDEML.LIB
DdePostAdvise	DDEML	DDEML.LIB
DdeQueryConvInfo	DDEML	DDEML.LIB
DdeQueryNextServer	DDEML	DDEML.LIB
DdeQueryString	DDEML	DDEML.LIB
DdeReconnect	DDEML	DDEML.LIB
DdeSetUserHandle	DDEML	DDEML.LIB
DdeUnaccessData	DDEML	DDEML.LIB
DdeUninitialize	DDEML	DDEML.LIB
DebugBreak	KERNEL	LIBW.LIB
DebugOutput	KERNEL	LIBW.LIB
DefDlgProc	USER	LIBW.LIB
DefDriverProc	USER	LIBW.LIB
DeferWindowPos	USER	LIBW.LIB
DefFrameProc	USER	LIBW.LIB
DefHookProc	USER	LIBW.LIB
DefMDIChildProc	USER	LIBW.LIB
DefScreenSaverProc	—	SCRNSAVE.LIB
DefWindowProc	USER	LIBW.LIB
DeleteAtom	KERNEL	LIBW.LIB
DeleteDC	GDI	LIBW.LIB
DeleteMenu	USER	LIBW.LIB
DeleteMetaFile	GDI	LIBW.LIB
DeleteObject	GDI	LIBW.LIB
DestroyCaret	USER	LIBW.LIB
DestroyCursor	USER	LIBW.LIB
DestroyIcon	USER	LIBW.LIB
DestroyMenu	USER	LIBW.LIB
DestroyWindow	USER	LIBW.LIB
DialogBox	USER	LIBW.LIB

Function	Module	Import library
DialogBoxIndirect	USER	LIBW.LIB
DialogBoxIndirectParam	USER	LIBW.LIB
DialogBoxParam	USER	LIBW.LIB
DirectedYield	KERNEL	LIBW.LIB
DispatchMessage	USER	LIBW.LIB
DlgChangePassword	—	SCRNSAVE.LIB
DlgDirList	USER	LIBW.LIB
DlgDirListComboBox	USER	LIBW.LIB
DlgDirSelect	USER	LIBW.LIB
DlgDirSelectComboBox	USER	LIBW.LIB
DlgDirSelectComboBoxEx	USER	LIBW.LIB
DlgDirSelectEx	USER	LIBW.LIB
DlgGetPassword	—	SCRNSAVE.LIB
DlgInvalidPassword	—	SCRNSAVE.LIB
DOS3Call	KERNEL	LIBW.LIB
DPtoLP	GDI	LIBW.LIB
DragAcceptFiles	SHELL	SHELL.LIB
DragFinish	SHELL	SHELL.LIB
DragQueryFile	SHELL	SHELL.LIB
DragQueryPoint	SHELL	SHELL.LIB
DrawFocusRect	USER	LIBW.LIB
DrawIcon	USER	LIBW.LIB
DrawMenuBar	USER	LIBW.LIB
DrawText	USER	LIBW.LIB
Ellipse	GDI	LIBW.LIB
EmptyClipboard	USER	LIBW.LIB
EnableCommNotification	USER	LIBW.LIB
EnableHardwareInput	USER	LIBW.LIB
EnableMenuItem	USER	LIBW.LIB
EnableScrollBar	USER	LIBW.LIB
EnableWindow	USER	LIBW.LIB
EndDeferWindowPos	USER	LIBW.LIB
EndDialog	USER	LIBW.LIB
EndDoc	GDI	LIBW.LIB
EndPage	GDI	LIBW.LIB
EndPaint	USER	LIBW.LIB
EnumChildWindows	USER	LIBW.LIB
EnumClipboardFormats	USER	LIBW.LIB

Function	Module	Import library
EnumFontFamilies	GDI	LIBW.LIB
EnumFonts	GDI	LIBW.LIB
EnumMetaFile	GDI	LIBW.LIB
EnumObjects	GDI	LIBW.LIB
EnumProps	USER	LIBW.LIB
EnumTaskWindows	USER	LIBW.LIB
EnumWindows	USER	LIBW.LIB
EqualRect	USER	LIBW.LIB
EqualRgn	GDI	LIBW.LIB
Escape	GDI	LIBW.LIB
EscapeCommFunction	USER	LIBW.LIB
ExcludeClipRect	GDI	LIBW.LIB
ExcludeUpdateRgn	USER	LIBW.LIB
ExitWindows	USER	LIBW.LIB
ExitWindowsExec	USER	LIBW.LIB
ExtFloodFill	GDI	LIBW.LIB
ExtractIcon	SHELL	SHELL.LIB
ExtTextOut	GDI	LIBW.LIB
FatalAppExit	KERNEL	LIBW.LIB
FatalExit	KERNEL	LIBW.LIB
FillRect	USER	LIBW.LIB
FillRgn	GDI	LIBW.LIB
FindAtom	KERNEL	LIBW.LIB
FindExecutable	SHELL	SHELL.LIB
FindResource	KERNEL	LIBW.LIB
FindText	COMMDLG	COMMDLG.LIB
FindWindow	USER	LIBW.LIB
FlashWindow	USER	LIBW.LIB
FloodFill	GDI	LIBW.LIB
FlushComm	USER	LIBW.LIB
FrameRect	USER	LIBW.LIB
FrameRgn	GDI	LIBW.LIB
FreeAllGDIMem	STRESS	STRESS.LIB
FreeAllMem	STRESS	STRESS.LIB
FreeAllUserMem	STRESS	STRESS.LIB
FreeLibrary	KERNEL	LIBW.LIB
FreeModule	KERNEL	LIBW.LIB
FreeProcInstance	KERNEL	LIBW.LIB

Function	Module	Import library
FreeResource	KERNEL	LIBW.LIB
FreeSelector	KERNEL	LIBW.LIB
GetActiveWindow	USER	LIBW.LIB
GetAspectRatioFilter	GDI	LIBW.LIB
GetAspectRatioFilterEx	GDI	LIBW.LIB
GetAsyncKeyState	USER	LIBW.LIB
GetAtomHandle	KERNEL	LIBW.LIB
GetAtomName	KERNEL	LIBW.LIB
GetBitmapBits	GDI	LIBW.LIB
GetBitmapDimension	GDI	LIBW.LIB
GetBitmapDimensionEx	GDI	LIBW.LIB
GetBkColor	GDI	LIBW.LIB
GetBkMode	GDI	LIBW.LIB
GetBoundsRect	GDI	LIBW.LIB
GetBrushOrg	GDI	LIBW.LIB
GetBrushOrgEx	GDI	LIBW.LIB
GetCapture	USER	LIBW.LIB
GetCaretBlinkTime	USER	LIBW.LIB
GetCaretPos	USER	LIBW.LIB
GetCharABCWidths	GDI	LIBW.LIB
GetCharWidth	GDI	LIBW.LIB
GetClassInfo	USER	LIBW.LIB
GetClassLong	USER	LIBW.LIB
GetClassName	USER	LIBW.LIB
GetClassWord	USER	LIBW.LIB
GetClientRect	USER	LIBW.LIB
GetClipboardData	USER	LIBW.LIB
GetClipboardFormatName	USER	LIBW.LIB
GetClipboardOwner	USER	LIBW.LIB
GetClipboardViewer	USER	LIBW.LIB
GetClipBox	GDI	LIBW.LIB
GetClipCursor	USER	LIBW.LIB
GetCodeHandle	KERNEL	LIBW.LIB
GetCodeInfo	KERNEL	LIBW.LIB
GetCommError	USER	LIBW.LIB
GetCommEventMask	USER	LIBW.LIB
GetCommState	USER	LIBW.LIB
GetCurrentPDB	KERNEL	LIBW.LIB

Function	Module	Import library
GetCurrentPosition	GDI	LIBW.LIB
GetCurrentPositionEx	GDI	LIBW.LIB
GetCurrentTask	KERNEL	LIBW.LIB
GetCurrentTime	USER	LIBW.LIB
GetCursor	USER	LIBW.LIB
GetCursorPos	USER	LIBW.LIB
GetDC	USER	LIBW.LIB
GetDCEx	USER	LIBW.LIB
GetDCOrg	GDI	LIBW.LIB
GetDesktopWindow	USER	LIBW.LIB
GetDeviceCaps	GDI	LIBW.LIB
GetDialogBaseUnits	USER	LIBW.LIB
GetDIBits	GDI	LIBW.LIB
GetDlgCtrlID	USER	LIBW.LIB
GetDlgItem	USER	LIBW.LIB
GetDlgItemInt	USER	LIBW.LIB
GetDlgItemText	USER	LIBW.LIB
GetDOSEnvironment	KERNEL	LIBW.LIB
GetDoubleClickTime	USER	LIBW.LIB
GetDriverInfo	USER	LIBW.LIB
GetDriverModuleHandle	USER	LIBW.LIB
GetDriveType	KERNEL	LIBW.LIB
GetExpandedName	LZEXPAND	LZEXPAND.LIB
GetFileResource	VER	VER.LIB
GetFileResourceSize	VER	VER.LIB
GetFileTitle	COMMDLG	COMMDLG.LIB
GetFileVersionInfo	VER	VER.LIB
GetFileVersionInfoSize	VER	VER.LIB
GetFocus	USER	LIBW.LIB
GetFontData	GDI	LIBW.LIB
GetFreeFileHandles	STRESS	STRESS.LIB
GetFreeSpace	KERNEL	LIBW.LIB
GetFreeSystemResources	USER	LIBW.LIB
GetGlyphOutline	GDI	LIBW.LIB
GetInputState	USER	LIBW.LIB
GetInstanceData	KERNEL	LIBW.LIB
GetKBCodePage	KEYBOARD	LIBW.LIB
GetKerningPairs	GDI	LIBW.LIB

Function	Module	Import library
GetKeyboardState	USER	LIBW.LIB
GetKeyboardType	KEYBOARD	LIBW.LIB
GetKeyNameText	KEYBOARD	LIBW.LIB
GetKeyState	USER	LIBW.LIB
GetLastActivePopup	USER	LIBW.LIB
GetMapMode	GDI	LIBW.LIB
GetMenu	USER	LIBW.LIB
GetMenuCheckMarkDimensions	USER	LIBW.LIB
GetMenuItemCount	USER	LIBW.LIB
GetMenuItemID	USER	LIBW.LIB
GetMenuState	USER	LIBW.LIB
GetMenuString	USER	LIBW.LIB
GetMessage	USER	LIBW.LIB
GetMessageExtraInfo	USER	LIBW.LIB
GetMessagePos	USER	LIBW.LIB
GetMessageTime	USER	LIBW.LIB
GetMetaFile	GDI	LIBW.LIB
GetMetaFileBits	GDI	LIBW.LIB
GetModuleFileName	KERNEL	LIBW.LIB
GetModuleHandle	KERNEL	LIBW.LIB
GetModuleUsage	KERNEL	LIBW.LIB
GetNearestColor	GDI	LIBW.LIB
GetNearestPaletteIndex	GDI	LIBW.LIB
GetNextDlgGroupItem	USER	LIBW.LIB
GetNextDlgTabItem	USER	LIBW.LIB
GetNextDriver	USER	LIBW.LIB
GetNextWindow	USER	LIBW.LIB
GetNumTasks	KERNEL	LIBW.LIB
GetObject	GDI	LIBW.LIB
GetOpenClipboardWindow	USER	LIBW.LIB
GetOpenFileName	COMMDLG	COMMDLG.LIB
GetOutlineTextMetrics	GDI	LIBW.LIB
GetPaletteEntries	GDI	LIBW.LIB
GetParent	USER	LIBW.LIB
GetPixel	GDI	LIBW.LIB
GetPolyFillMode	GDI	LIBW.LIB
GetPriorityClipboardFormat	USER	LIBW.LIB
GetPrivateProfileInt	KERNEL	LIBW.LIB

Function	Module	Import library
GetPrivateProfileString	KERNEL	LIBW.LIB
GetProcAddress	KERNEL	LIBW.LIB
GetProfileInt	KERNEL	LIBW.LIB
GetProfileString	KERNEL	LIBW.LIB
GetProp	USER	LIBW.LIB
GetQueueStatus	USER	LIBW.LIB
GetRasterizerCaps	GDI	LIBW.LIB
GetRgnBox	GDI	LIBW.LIB
GetROP2	GDI	LIBW.LIB
GetSaveFileName	COMMDLG	COMMDLG.LIB
GetScrollPos	USER	LIBW.LIB
GetScrollRange	USER	LIBW.LIB
GetSelectorBase	KERNEL	LIBW.LIB
GetSelectorLimit	KERNEL	LIBW.LIB
GetStockObject	GDI	LIBW.LIB
GetStretchBltMode	GDI	LIBW.LIB
GetSubMenu	USER	LIBW.LIB
GetSysColor	USER	LIBW.LIB
GetSysModalWindow	USER	LIBW.LIB
GetSystemDebugState	USER	LIBW.LIB
GetSystemDir	—	VERS.LIB
GetSystemDirectory	KERNEL	LIBW.LIB
GetSystemMenu	USER	LIBW.LIB
GetSystemMetrics	USER	LIBW.LIB
GetSystemPaletteEntries	GDI	LIBW.LIB
GetSystemPaletteUse	GDI	LIBW.LIB
GetTabbedTextExtent	USER	LIBW.LIB
GetTempDrive	KERNEL	LIBW.LIB
GetTempFileName	KERNEL	LIBW.LIB
GetTextAlign	GDI	LIBW.LIB
GetTextCharacterExtra	GDI	LIBW.LIB
GetTextColor	GDI	LIBW.LIB
GetTextExtent	GDI	LIBW.LIB
GetTextExtentPoint	GDI	LIBW.LIB
GetTextFace	GDI	LIBW.LIB
GetTextMetrics	GDI	LIBW.LIB
GetTickCount	USER	LIBW.LIB
GetTimerResolution	USER	LIBW.LIB

Function	Module	Import library
GetTopWindow	USER	LIBW.LIB
GetUpdateRect	USER	LIBW.LIB
GetUpdateRgn	USER	LIBW.LIB
GetVersion	KERNEL	LIBW.LIB
GetViewportExt	GDI	LIBW.LIB
GetViewportExtEx	GDI	LIBW.LIB
GetViewportOrg	GDI	LIBW.LIB
GetViewportOrgEx	GDI	LIBW.LIB
GetWinDebugInfo	KERNEL	LIBW.LIB
GetWindow	USER	LIBW.LIB
GetWindowDC	USER	LIBW.LIB
GetWindowExt	GDI	LIBW.LIB
GetWindowExtEx	GDI	LIBW.LIB
GetWindowLong	USER	LIBW.LIB
GetWindowOrg	GDI	LIBW.LIB
GetWindowOrgEx	GDI	LIBW.LIB
GetWindowPlacement	USER	LIBW.LIB
GetWindowRect	USER	LIBW.LIB
GetWindowsDir	—	VERS.LIB
GetWindowsDirectory	KERNEL	LIBW.LIB
GetWindowTask	USER	LIBW.LIB
GetWindowText	USER	LIBW.LIB
GetWindowTextLength	USER	LIBW.LIB
GetWindowWord	USER	LIBW.LIB
GetWinFlags	KERNEL	LIBW.LIB
GetWinMem32Version	WINMEM32	WINMEM32.LIB
Global16PointerAlloc	WINMEM32	WINMEM32.LIB
Global16PointerFree	WINMEM32	WINMEM32.LIB
Global32Alloc	WINMEM32	WINMEM32.LIB
Global32CodeAlias	WINMEM32	WINMEM32.LIB
Global32CodeAliasFree	WINMEM32	WINMEM32.LIB
Global32Free	WINMEM32	WINMEM32.LIB
Global32Realloc	WINMEM32	WINMEM32.LIB
GlobalAddAtom	USER	LIBW.LIB
GlobalAlloc	KERNEL	LIBW.LIB
GlobalCompact	KERNEL	LIBW.LIB
GlobalDeleteAtom	USER	LIBW.LIB
GlobalDosAlloc	KERNEL	LIBW.LIB

Function	Module	Import library
GlobalDosFree	KERNEL	LIBW.LIB
GlobalEntryHandle	TOOLHELP	TOOLHELP.LIB
GlobalEntryModule	TOOLHELP	TOOLHELP.LIB
GlobalFindAtom	USER	LIBW.LIB
GlobalFirst	TOOLHELP	TOOLHELP.LIB
GlobalFix	KERNEL	LIBW.LIB
GlobalFlags	KERNEL	LIBW.LIB
GlobalFree	KERNEL	LIBW.LIB
GlobalGetAtomName	USER	LIBW.LIB
GlobalHandle	KERNEL	LIBW.LIB
GlobalHandleToSel	TOOLHELP	TOOLHELP.LIB
GlobalInfo	TOOLHELP	TOOLHELP.LIB
GlobalLock	KERNEL	LIBW.LIB
GlobalLRUNewest	KERNEL	LIBW.LIB
GlobalLRUOldest	KERNEL	LIBW.LIB
GlobalNext	TOOLHELP	TOOLHELP.LIB
GlobalNotify	KERNEL	LIBW.LIB
GlobalPageLock	KERNEL	LIBW.LIB
GlobalPageUnlock	KERNEL	LIBW.LIB
GlobalReAlloc	KERNEL	LIBW.LIB
GlobalSize	KERNEL	LIBW.LIB
GlobalUnfix	KERNEL	LIBW.LIB
GlobalUnlock	KERNEL	LIBW.LIB
GlobalUnWire	KERNEL	LIBW.LIB
GlobalWire	KERNEL	LIBW.LIB
GrayString	USER	LIBW.LIB
hardware_event	USER	LIBW.LIB
HelpMessageFilterHookFunction	—	SCRNSAVE.LIB
HideCaret	USER	LIBW.LIB
HiliteMenuItem	USER	LIBW.LIB
hmemcpy	KERNEL	LIBW.LIB
_hread	KERNEL	LIBW.LIB
_hwrite	KERNEL	LIBW.LIB
InflateRect	USER	LIBW.LIB
InitAtomTable	KERNEL	LIBW.LIB
InSendMessage	USER	LIBW.LIB
InsertMenu	USER	LIBW.LIB
InterruptRegister	TOOLHELP	TOOLHELP.LIB

Function	Module	Import library
InterruptUnRegister	TOOLHELP	TOOLHELP.LIB
IntersectClipRect	GDI	LIBW.LIB
IntersectRect	USER	LIBW.LIB
InvalidateRect	USER	LIBW.LIB
InvalidateRgn	USER	LIBW.LIB
InvertRect	USER	LIBW.LIB
InvertRgn	GDI	LIBW.LIB
IsBadCodePtr	KERNEL	LIBW.LIB
IsBadHugeReadPtr	KERNEL	LIBW.LIB
IsBadHugeWritePtr	KERNEL	LIBW.LIB
IsBadReadPtr	KERNEL	LIBW.LIB
IsBadStringPtr	KERNEL	LIBW.LIB
IsBadWritePtr	KERNEL	LIBW.LIB
IsCharAlpha	USER	LIBW.LIB
IsCharAlphaNumeric	USER	LIBW.LIB
IsCharLower	USER	LIBW.LIB
IsCharUpper	USER	LIBW.LIB
IsChild	USER	LIBW.LIB
IsClipboardFormatAvailable	USER	LIBW.LIB
IsDBCSLeadByte	KERNEL	LIBW.LIB
IsDialogMessage	USER	LIBW.LIB
IsDlgButtonChecked	USER	LIBW.LIB
IsGDIObject	GDI	LIBW.LIB
IsIconic	USER	LIBW.LIB
IsMenu	USER	LIBW.LIB
IsRectEmpty	USER	LIBW.LIB
IsTask	KERNEL	LIBW.LIB
IsWindow	USER	LIBW.LIB
IsWindowEnabled	USER	LIBW.LIB
IsWindowVisible	USER	LIBW.LIB
IsZoomed	USER	LIBW.LIB
KillTimer	USER	LIBW.LIB
_lclose	KERNEL	LIBW.LIB
_lcreat	KERNEL	LIBW.LIB
LimitEmsPages	KERNEL	LIBW.LIB
LineDDA	GDI	LIBW.LIB
LineTo	GDI	LIBW.LIB
_llseek	KERNEL	LIBW.LIB

Function	Module	Import library
LoadAccelerators	USER	LIBW.LIB
LoadBitmap	USER	LIBW.LIB
LoadCursor	USER	LIBW.LIB
LoadIcon	USER	LIBW.LIB
LoadLibrary	KERNEL	LIBW.LIB
LoadMenu	USER	LIBW.LIB
LoadMenuIndirect	USER	LIBW.LIB
LoadModule	KERNEL	LIBW.LIB
LoadResource	KERNEL	LIBW.LIB
LoadString	USER	LIBW.LIB
LocalAlloc	KERNEL	LIBW.LIB
LocalCompact	KERNEL	LIBW.LIB
LocalFirst	TOOLHELP	TOOLHELP.LIB
LocalFlags	KERNEL	LIBW.LIB
LocalFree	KERNEL	LIBW.LIB
LocalHandle	KERNEL	LIBW.LIB
LocalInfo	TOOLHELP	TOOLHELP.LIB
LocalInit	KERNEL	LIBW.LIB
LocalLock	KERNEL	LIBW.LIB
LocalNext	TOOLHELP	TOOLHELP.LIB
LocalReAlloc	KERNEL	LIBW.LIB
LocalShrink	KERNEL	LIBW.LIB
LocalSize	KERNEL	LIBW.LIB
LocalUnlock	KERNEL	LIBW.LIB
LockInput	USER	LIBW.LIB
LockResource	KERNEL	LIBW.LIB
LockSegment	KERNEL	LIBW.LIB
LockWindowUpdate	USER	LIBW.LIB
LogError	KERNEL	LIBW.LIB
LogParamError	KERNEL	LIBW.LIB
_lopen	KERNEL	LIBW.LIB
LPtoDP	GDI	LIBW.LIB
_lread	KERNEL	LIBW.LIB
lstrcat	KERNEL	LIBW.LIB
lstrcmp	USER	LIBW.LIB
lstrcmpi	USER	LIBW.LIB
lstrcpy	KERNEL	LIBW.LIB
lstrlen	KERNEL	LIBW.LIB

Function	Module	Import library
_lwrite	KERNEL	LIBW.LIB
LZClose	LZEXPAND	LZEXPAND.LIB
LZCopy	LZEXPAND	LZEXPAND.LIB
LZDone	LZEXPAND	LZEXPAND.LIB
LZInit	LZEXPAND	LZEXPAND.LIB
LZOpenFile	LZEXPAND	LZEXPAND.LIB
LZRead	LZEXPAND	LZEXPAND.LIB
LZSeek	LZEXPAND	LZEXPAND.LIB
LZStart	LZEXPAND	LZEXPAND.LIB
MakeProcInstance	KERNEL	LIBW.LIB
MapDialogRect	USER	LIBW.LIB
MapVirtualKey	KEYBOARD	LIBW.LIB
MapWindowPoints	USER	LIBW.LIB
MemManInfo	TOOLHELP	TOOLHELP.LIB
MemoryRead	TOOLHELP	TOOLHELP.LIB
MemoryWrite	TOOLHELP	TOOLHELP.LIB
MessageBeep	USER	LIBW.LIB
MessageBox	USER	LIBW.LIB
ModifyMenu	USER	LIBW.LIB
ModuleFindHandle	TOOLHELP	TOOLHELP.LIB
ModuleFindName	TOOLHELP	TOOLHELP.LIB
ModuleFirst	TOOLHELP	TOOLHELP.LIB
ModuleNext	TOOLHELP	TOOLHELP.LIB
MoveTo	GDI	LIBW.LIB
MoveToEx	GDI	LIBW.LIB
MoveWindow	USER	LIBW.LIB
MulDiv	GDI	LIBW.LIB
NetBIOSCall	KERNEL	LIBW.LIB
NotifyRegister	TOOLHELP	TOOLHELP.LIB
NotifyUnRegister	TOOLHELP	TOOLHELP.LIB
OemKeyScan	KEYBOARD	LIBW.LIB
OemToAnsi	KEYBOARD	LIBW.LIB
OemToAnsiBuff	KEYBOARD	LIBW.LIB
OffsetClipRgn	GDI	LIBW.LIB
OffsetRect	USER	LIBW.LIB
OffsetRgn	GDI	LIBW.LIB
OffsetViewportOrg	GDI	LIBW.LIB
OffsetViewportOrgEx	GDI	LIBW.LIB

Function	Module	Import library
OffsetWindowOrg	GDI	LIBW.LIB
OffsetWindowOrgEx	GDI	LIBW.LIB
OleActivate	OLECLI	OLECLI.LIB
OleBlockServer	OLESVR	OLESVR.LIB
OleClone	OLECLI	OLECLI.LIB
OleClose	OLECLI	OLECLI.LIB
OleCopyFromLink	OLECLI	OLECLI.LIB
OleCopyToClipboard	OLECLI	OLECLI.LIB
OleCreate	OLECLI	OLECLI.LIB
OleCreateFromClip	OLECLI	OLECLI.LIB
OleCreateFromFile	OLECLI	OLECLI.LIB
OleCreateFromTemplate	OLECLI	OLECLI.LIB
OleCreateInvisible	OLECLI	OLECLI.LIB
OleCreateLinkFromClip	OLECLI	OLECLI.LIB
OleCreateLinkFromFile	OLECLI	OLECLI.LIB
OleDelete	OLECLI	OLECLI.LIB
OleDraw	OLECLI	OLECLI.LIB
OleEnumFormats	OLECLI	OLECLI.LIB
OleEnumObjects	OLECLI	OLECLI.LIB
OleEqual	OLECLI	OLECLI.LIB
OleExecute	OLECLI	OLECLI.LIB
OleGetData	OLECLI	OLECLI.LIB
OleGetLinkUpdateOptions	OLECLI	OLECLI.LIB
OleIsDcMeta	OLECLI	OLECLI.LIB
OleLoadFromStream	OLECLI	OLECLI.LIB
OleLockServer	OLECLI	OLECLI.LIB
OleObjectConvert	OLECLI	OLECLI.LIB
OleQueryBounds	OLECLI	OLECLI.LIB
OleQueryClientVersion	OLECLI	OLECLI.LIB
OleQueryCreateFromClip	OLECLI	OLECLI.LIB
OleQueryLinkFromClip	OLECLI	OLECLI.LIB
OleQueryName	OLECLI	OLECLI.LIB
OleQueryOpen	OLECLI	OLECLI.LIB
OleQueryOutOfDate	OLECLI	OLECLI.LIB
OleQueryProtocol	OLECLI	OLECLI.LIB
OleQueryReleaseError	OLECLI	OLECLI.LIB
OleQueryReleaseMethod	OLECLI	OLECLI.LIB
OleQueryReleaseStatus	OLECLI	OLECLI.LIB

Function	Module	Import library
OleQueryServerVersion	OLESVR	OLESVR.LIB
OleQuerySize	OLECLI	OLECLI.LIB
OleQueryType	OLECLI	OLECLI.LIB
OleReconnect	OLECLI	OLECLI.LIB
OleRegisterClientDoc	OLECLI	OLECLI.LIB
OleRegisterServer	OLESVR	OLESVR.LIB
OleRegisterServerDoc	OLESVR	OLESVR.LIB
OleRelease	OLECLI	OLECLI.LIB
OleRename	OLECLI	OLECLI.LIB
OleRenameClientDoc	OLECLI	OLECLI.LIB
OleRenameServerDoc	OLESVR	OLESVR.LIB
OleRequestData	OLECLI	OLECLI.LIB
OleRevertClientDoc	OLECLI	OLECLI.LIB
OleRevertServerDoc	OLESVR	OLESVR.LIB
OleRevokeClientDoc	OLECLI	OLECLI.LIB
OleRevokeObject	OLESVR	OLESVR.LIB
OleRevokeServer	OLESVR	OLESVR.LIB
OleRevokeServerDoc	OLESVR	OLESVR.LIB
OleSavedClientDoc	OLECLI	OLECLI.LIB
OleSavedServerDoc	OLESVR	OLESVR.LIB
OleSaveToStream	OLECLI	OLECLI.LIB
OleSetBounds	OLECLI	OLECLI.LIB
OleSetColorScheme	OLECLI	OLECLI.LIB
OleSetData	OLECLI	OLECLI.LIB
OleSetHostNames	OLECLI	OLECLI.LIB
OleSetLinkUpdateOptions	OLECLI	OLECLI.LIB
OleSetTargetDevice	OLECLI	OLECLI.LIB
OleUnblockServer	OLESVR	OLESVR.LIB
OleUnlockServer	OLECLI	OLECLI.LIB
OleUpdate	OLECLI	OLECLI.LIB
OpenClipboard	USER	LIBW.LIB
OpenComm	USER	LIBW.LIB
OpenDriver	USER	LIBW.LIB
OpenFile	LZEXPAND	LZEXPAND.LIB
OpenIcon	USER	LIBW.LIB
OutputDebugString	KERNEL	LIBW.LIB
PaintRgn	GDI	LIBW.LIB
PatBlt	GDI	LIBW.LIB

Function	Module	Import library
PeekMessage	USER	LIBW.LIB
Pie	GDI	LIBW.LIB
PlayMetaFile	GDI	LIBW.LIB
PlayMetaFileRecord	GDI	LIBW.LIB
Polygon	GDI	LIBW.LIB
Polyline	GDI	LIBW.LIB
PolyPolygon	GDI	LIBW.LIB
PostAppMessage	USER	LIBW.LIB
PostMessage	USER	LIBW.LIB
PostQuitMessage	USER	LIBW.LIB
PrestoChangoSelector	KERNEL	LIBW.LIB
PrintDlg	COMMDLG	COMMDLG.LIB
ProfClear	—	LIBW.LIB
ProfFinish	—	LIBW.LIB
ProfFlush	—	LIBW.LIB
ProfInsChk	—	LIBW.LIB
ProfSampRate	—	LIBW.LIB
ProfSetup	—	LIBW.LIB
ProfStart	—	LIBW.LIB
ProfStop	—	LIBW.LIB
PtInRect	USER	LIBW.LIB
PtInRegion	GDI	LIBW.LIB
PtVisible	GDI	LIBW.LIB
QueryAbort	GDI	LIBW.LIB
QuerySendMessage	USER	LIBW.LIB
ReadComm	USER	LIBW.LIB
RealizePalette	USER	LIBW.LIB
Rectangle	GDI	LIBW.LIB
RectInRegion	GDI	LIBW.LIB
RectVisible	GDI	LIBW.LIB
RedrawWindow	USER	LIBW.LIB
RegCloseKey	SHELL	SHELL.LIB
RegCreateKey	SHELL	SHELL.LIB
RegDeleteKey	SHELL	SHELL.LIB
RegEnumKey	SHELL	SHELL.LIB
RegisterClass	USER	LIBW.LIB
RegisterClipboardFormat	USER	LIBW.LIB
RegisterWindowMessage	USER	LIBW.LIB

Function	Module	Import library
RegOpenKey	SHELL	SHELL.LIB
RegQueryValue	SHELL	SHELL.LIB
RegSetValue	SHELL	SHELL.LIB
ReleaseCapture	USER	LIBW.LIB
ReleaseDC	USER	LIBW.LIB
RemoveFontResource	GDI	LIBW.LIB
RemoveMenu	USER	LIBW.LIB
RemoveProp	USER	LIBW.LIB
ReplaceText	COMMDLG	COMMDLG.LIB
ReplyMessage	USER	LIBW.LIB
ResetDC	GDI	LIBW.LIB
ResizePalette	GDI	LIBW.LIB
RestoreDC	GDI	LIBW.LIB
RoundRect	GDI	LIBW.LIB
SaveDC	GDI	LIBW.LIB
ScaleViewportExt	GDI	LIBW.LIB
ScaleViewportExtEx	GDI	LIBW.LIB
ScaleWindowExt	GDI	LIBW.LIB
ScaleWindowExtEx	GDI	LIBW.LIB
ScreenSaverProc	—	SCRNSAVE.LIB
ScreenToClient	USER	LIBW.LIB
ScrollDC	USER	LIBW.LIB
ScrollWindow	USER	LIBW.LIB
ScrollWindowEx	USER	LIBW.LIB
SelectClipRgn	GDI	LIBW.LIB
SelectObject	GDI	LIBW.LIB
SelectPalette	USER	LIBW.LIB
SendDlgItemMessage	USER	LIBW.LIB
SendDriverMessage	USER	LIBW.LIB
SendMessage	USER	LIBW.LIB
SetAbortProc	GDI	LIBW.LIB
SetActiveWindow	USER	LIBW.LIB
SetBitmapBits	GDI	LIBW.LIB
SetBitmapDimension	GDI	LIBW.LIB
SetBitmapDimensionEx	GDI	LIBW.LIB
SetBkColor	GDI	LIBW.LIB
SetBkMode	GDI	LIBW.LIB
SetBoundsRect	GDI	LIBW.LIB

Function	Module	Import library
SetBrushOrg	GDI	LIBW.LIB
SetCapture	USER	LIBW.LIB
SetCaretBlinkTime	USER	LIBW.LIB
SetCaretPos	USER	LIBW.LIB
SetClassLong	USER	LIBW.LIB
SetClassWord	USER	LIBW.LIB
SetClipboardData	USER	LIBW.LIB
SetClipboardViewer	USER	LIBW.LIB
SetCommBreak	USER	LIBW.LIB
SetCommEventMask	USER	LIBW.LIB
SetCommState	USER	LIBW.LIB
SetCursor	USER	LIBW.LIB
SetCursorPos	USER	LIBW.LIB
SetDIBits	GDI	LIBW.LIB
SetDIBitsToDevice	GDI	LIBW.LIB
SetDlgItemInt	USER	LIBW.LIB
SetDlgItemText	USER	LIBW.LIB
SetDoubleClickTime	USER	LIBW.LIB
SetErrorMode	KERNEL	LIBW.LIB
SetFocus	USER	LIBW.LIB
SetHandleCount	KERNEL	LIBW.LIB
SetKeyboardState	USER	LIBW.LIB
SetMapMode	GDI	LIBW.LIB
SetMapperFlags	GDI	LIBW.LIB
SetMenu	USER	LIBW.LIB
SetMenuItemBitmaps	USER	LIBW.LIB
SetMessageQueue	USER	LIBW.LIB
SetMetaFileBits	GDI	LIBW.LIB
SetMetaFileBitsBetter	GDI	LIBW.LIB
SetPaletteEntries	GDI	LIBW.LIB
SetParent	USER	LIBW.LIB
SetPixel	GDI	LIBW.LIB
SetPolyFillMode	GDI	LIBW.LIB
SetProp	USER	LIBW.LIB
SetRect	USER	LIBW.LIB
SetRectEmpty	USER	LIBW.LIB
SetRectRgn	GDI	LIBW.LIB
SetResourceHandler	KERNEL	LIBW.LIB

Function	Module	Import library
SetROP2	GDI	LIBW.LIB
SetScrollPos	USER	LIBW.LIB
SetScrollRange	USER	LIBW.LIB
SetSelectorBase	KERNEL	LIBW.LIB
SetSelectorLimit	KERNEL	LIBW.LIB
SetStretchBltMode	GDI	LIBW.LIB
SetSwapAreaSize	KERNEL	LIBW.LIB
SetSysColors	USER	LIBW.LIB
SetSysModalWindow	USER	LIBW.LIB
SetSystemPaletteUse	GDI	LIBW.LIB
SetTextAlign	GDI	LIBW.LIB
SetTextCharacterExtra	GDI	LIBW.LIB
SetTextColor	GDI	LIBW.LIB
SetTextJustification	GDI	LIBW.LIB
SetTimer	USER	LIBW.LIB
SetViewportExt	GDI	LIBW.LIB
SetViewportExtEx	GDI	LIBW.LIB
SetViewportOrg	GDI	LIBW.LIB
SetViewportOrgEx	GDI	LIBW.LIB
SetWinDebugInfo	KERNEL	LIBW.LIB
SetWindowExt	GDI	LIBW.LIB
SetWindowExtEx	GDI	LIBW.LIB
SetWindowLong	USER	LIBW.LIB
SetWindowOrg	GDI	LIBW.LIB
SetWindowOrgEx	GDI	LIBW.LIB
SetWindowPlacement	USER	LIBW.LIB
SetWindowPos	USER	LIBW.LIB
SetWindowsHook	USER	LIBW.LIB
SetWindowsHookEx	USER	LIBW.LIB
SetWindowText	USER	LIBW.LIB
SetWindowWord	USER	LIBW.LIB
ShellExecute	SHELL	SHELL.LIB
ShowCaret	USER	LIBW.LIB
ShowCursor	USER	LIBW.LIB
ShowOwnedPopups	USER	LIBW.LIB
ShowScrollBar	USER	LIBW.LIB
ShowWindow	USER	LIBW.LIB
SizeofResource	KERNEL	LIBW.LIB

Function	Module	Import library
SpoolFile	GDI	LIBW.LIB
StackTraceCSIPFirst	TOOLHELP	TOOLHELP.LIB
StackTraceFirst	TOOLHELP	TOOLHELP.LIB
StackTraceNext	TOOLHELP	TOOLHELP.LIB
StartDoc	GDI	LIBW.LIB
StartPage	GDI	LIBW.LIB
StretchBlt	GDI	LIBW.LIB
StretchDIBits	GDI	LIBW.LIB
SubtractRect	USER	LIBW.LIB
SwapMouseButton	USER	LIBW.LIB
SwapRecording	KERNEL	LIBW.LIB
SwitchStackBack	KERNEL	LIBW.LIB
SwitchStackTo	KERNEL	LIBW.LIB
SystemHeapInfo	TOOLHELP	TOOLHELP.LIB
SystemParametersInfo	USER	LIBW.LIB
TabbedTextOut	USER	LIBW.LIB
TaskFindHandle	TOOLHELP	TOOLHELP.LIB
TaskFirst	TOOLHELP	TOOLHELP.LIB
TaskGetCSIP	TOOLHELP	TOOLHELP.LIB
TaskNext	TOOLHELP	TOOLHELP.LIB
TaskSetCSIP	TOOLHELP	TOOLHELP.LIB
TaskSwitch	TOOLHELP	TOOLHELP.LIB
TerminateApp	TOOLHELP	TOOLHELP.LIB
TextOut	GDI	LIBW.LIB
Throw	KERNEL	LIBW.LIB
TimerCount	TOOLHELP	TOOLHELP.LIB
ToAscii	KEYBOARD	LIBW.LIB
TrackPopupMenu	USER	LIBW.LIB
TranslateAccelerator	USER	LIBW.LIB
TranslateMDISysAccel	USER	LIBW.LIB
TranslateMessage	USER	LIBW.LIB
TransmitCommChar	USER	LIBW.LIB
UnAllocDiskSpace	STRESS	STRESS.LIB
UnAllocFileHandles	STRESS	STRESS.LIB
UngetCommChar	USER	LIBW.LIB
UnhookWindowsHook	USER	LIBW.LIB
UnhookWindowsHookEx	USER	LIBW.LIB
UnionRect	USER	LIBW.LIB

Function	Module	Import library
UnlockSegment	KERNEL	LIBW.LIB
UnrealizeObject	GDI	LIBW.LIB
UnregisterClass	USER	LIBW.LIB
UpdateColors	GDI	LIBW.LIB
UpdateWindow	USER	LIBW.LIB
ValidateCodeSegments	KERNEL	LIBW.LIB
ValidateFreeSpaces	KERNEL	LIBW.LIB
ValidateRect	USER	LIBW.LIB
ValidateRgn	USER	LIBW.LIB
VerFindFile	VER	VER.LIB
VerInstallFile	VER	VER.LIB
VerLanguageName	VER	VER.LIB
VerQueryValue	VER	VER.LIB
VkKeyScan	KEYBOARD	LIBW.LIB
WaitMessage	USER	LIBW.LIB
WindowFromPoint	USER	LIBW.LIB
WinExec	KERNEL	LIBW.LIB
WinHelp	USER	LIBW.LIB
WNetAddConnection	USER	LIBW.LIB
WNetCancelConnection	USER	LIBW.LIB
WNetGetConnection	USER	LIBW.LIB
WriteComm	USER	LIBW.LIB
WritePrivateProfileString	KERNEL	LIBW.LIB
WriteProfileString	KERNEL	LIBW.LIB
_wsprintf	USER	LIBW.LIB
wvsprintf	USER	LIBW.LIB
Yield	KERNEL	LIBW.LIB

Index

Microsoft®

Microsoft® Windows™ 3.1 Programmer's Reference

Volume 1
Overview

This series of six volumes—the most accurate and up-to-date information on the Microsoft Windows operating system available anywhere—is the core documentation for the Microsoft Windows 3.1 Software Development Kit (SDK). Now updated to cover the Windows operating system version 3.1, the books contain information on all the new functions and services in the Microsoft Windows application programming interface (API), including new font management, application communication, and application integration capabilities. Look for all six titles in the *Microsoft Windows 3.1 Programmer's Reference Library*.

Microsoft Windows 3.1 Guide to Programming. A helpful introduction to the Windows API in version 3.1 for the experienced C programmer. Detailed instruction and examples. Topics include processing input and output, creating the necessary components of a Windows-based application, managing memory, using dynamic-link libraries and dynamic data exchange, and working with fonts and printers.

Microsoft Windows 3.1 Programmer's Reference, Volume 1: Overview. An examination of all the window management, graphics, and system services as well as the extension libraries that are part of the API. In addition, there is instruction on specific Windows-based applications for version 3.1: Control Panel, File Manager, and others.

Microsoft Windows 3.1 Programmer's Reference, Volume 2: Functions. A detailed reference to the API functions. Includes information on various function groups as well as an alphabetic reference to each function. Information includes syntax, statement of purpose, input parameters, return values, and comments.

Microsoft Windows 3.1 Programmer's Reference, Volume 3: Messages, Structures, and Macros. Comprehensive information on additional elements of the API: data types; structures; macros; printer escape codes; dynamic data exchange transactions; and File Manager, Control Panel, common dialog box, and installable driver messages.

Microsoft Windows 3.1 Programmer's Reference, Volume 4: Resources. Information on the many Windows file formats in version 3.1 as well as reference pages for several built-in tools. Reference-page topics include resource-definition statements, assembly-language macros, and Windows Help statements and macros.

Microsoft Windows 3.1 Programming Tools. Detailed information and instruction for using built-in software development tools that are part of the Microsoft Windows SDK; topics include creating and compiling resources, debugging applications, analyzing data, and compressing and decompressing data.

Please note: The six volumes of the *Microsoft Windows 3.1 Programmer's Reference Library* are included in the Microsoft Windows 3.1 Software Development Kit (SDK).

U.S.A.	**$29.95**
U.K.	£26.95
Canada	$39.95

[*Recommended*]

Microsoft PRESS®
The Authorized Editions

ISBN 1-55615-453-4
90000
9 781556 154539

Programming/Windows