

Microsoft® Windows

Software Development Kit

Programmer's Reference

Microsoft Corporation

Information in this document is subject to change without notice and does not represent a commitment on the part of Microsoft Corporation. The software described in this document is furnished under a license agreement or nondisclosure agreement. The software may be used or copied only in accordance with the terms of the agreement. It is against the law to copy this software on magnetic tape, disk, or any other medium for any purpose other than the purchaser's personal use.

© Copyright Microsoft Corporation, 1984, 1985, 1986

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

IBM is a registered trademark of International Business Machines Corporation.

Document Number 050051053-103-I01-1086
Part Number 050-150-056

Contents

About This Manual ix

1 Introduction 1

1.1	Introduction	3
1.2	Notational Conventions	3
1.3	Naming Conventions	4
1.4	Data Types	4
1.5	Windows Calling Convention	5
1.6	Near and Far Pointers	6

2 Window Functions 7

2.1	Introduction	9
2.2	Main Function	9
2.3	Message Functions	10
2.4	Window Function	20
2.5	Default Window Function	21
2.6	Window Class Functions	22
2.7	Window Creation Functions	26
2.8	Window Display and Movement Functions	32
2.9	Dialog Box Functions	39
2.10	Clipboard Functions	47
2.11	Input Functions	55
2.12	Menu Functions	61
2.13	Window Painting Functions	70
2.14	Scrolling Functions	77
2.15	Property List Functions	81
2.16	Window Attribute Functions	84
2.17	Error Functions	87
2.18	Cursor Functions	90
2.19	Caret Functions	92
2.20	Coordinate Functions	95
2.21	Rectangle Functions	97
2.22	System Information Functions	100
2.23	Window Hook Function	103

3 GDI Functions	105
3.1 Introduction	107
3.2 Display Context Functions	107
3.3 Output Functions	110
3.4 Drawing Object Functions	127
3.5 Selection Functions	138
3.6 Display Context Attribute Functions	140
3.7 Clipping Region Functions	159
3.8 Region Functions	162
3.9 Text Justification Functions	165
3.10 Metafile Functions	168
3.11 Control Functions	171
3.12 GDI Information Functions	180
3.13 Conversion Functions	187
4 System Resource Functions	189
4.1 Introduction	191
4.2 Module Manager Functions	191
4.3 Memory Manager Functions	196
4.4 Task Functions	206
4.5 Resource Manager Functions	207
4.6 String Translation Functions	217
4.7 Atom Manager Functions	219
4.8 Windows Initialization File Functions	222
4.9 Debugging Function	225
4.10 Communication Functions	225
4.11 Sound Functions	235
4.12 Utility Functions	242
4.13 File I/O Functions	245
5 Data Types and Structures	249
5.1 Introduction	251
5.2 Data Types	251
5.3 Window Data Structures	254
5.4 GDI Data Structures	259
5.5 Communication Data Structures	272
5.6 Open File Structure	277

6 File Formats 279

6.1	Introduction	281
6.2	Resource Script File	281
6.3	Module Definition File	311
6.4	Windows Initialization File	318

7 Assembly Language Macros 325

7.1	Introduction	327
7.2	CMACROS.INC File	327
7.3	Cmacros Options	327
7.4	Segment Macros	331
7.5	Using the Cmacros	346

8 Window Messages 349

8.1	Introduction	351
8.2	Window Management Messages	352
8.3	Initialization Messages	362
8.4	Input Messages	364
8.5	System Messages	377
8.6	Clipboard Messages	381
8.7	System Information Messages	386
8.8	Control Messages	388
8.9	Notification Messages	401
8.10	Scroll Bar Messages	402
8.11	Non-Client Area Messages	402

**A Raster Operation Codes
And Definitions 409**

A.1	Introduction	411
A.2	Operation Codes	412

B Virtual Key Code Summary 419

Contents

C Font Files 423

C.1	Introduction	425
C.2	Font File Formats	425
C.3	ANSI Character Set	432

Index 433

Tables

Table 2.1	Window Styles	28
Table 2.2	Virtual Keys	57
Table 2.3	ChangeMenu Flags	64
Table 3.1	Raster Operation Values	145
Table 3.2	GDI Functions and Metafiles	168
Table 3.3	GDI Information Indexes	183
Table 4.1	Communications Driver Error Codes	230
Table 6.1	Control Classes	300
Table 6.2	Control Styles	302
Table 7.1	Memory Options	328
Table 7.2	Calling Conventions	329
Table 7.3	Prolog/Epilog Code Options	330
Table C.1	ANSI Character Set for Windows	432

(

)

)

About This Manual

Purpose

This manual is an in-depth technical reference for system programmers creating Windows applications. This manual is intended to be used with the *Microsoft® Windows Programming Guide*. The guide explains the Windows system and illustrates how to create Windows applications.

It is assumed that all readers have some programming knowledge and experience.

Using This Manual

This manual contains eight chapters, three appendixes, and an index. Each part is briefly described below:

- Chapter 1 Provides an overview of this manual.
- Chapter 2 Describes the routines used to create and use windows.
- Chapter 3 Describes the Graphics Device Interface (GDI) routines.
- Chapter 4 Describes the routines used to access system resources.
- Chapter 5 Describes the data types and structures used by the Windows routines.
- Chapter 6 Describes the formats for input files used by Windows.
- Chapter 7 Describes the assembly language macros used to develop assembly language Windows applications.
- Chapter 8 Describes Windows messages.
- Appendix A Lists the raster operation codes and their definitions.
- Appendix B Lists the standard Windows key codes and their definitions.
- Appendix C Describes the font files and their format.

(

)

)

Chapter 1

Introduction

1.1	Introduction	3
1.2	Notational Conventions	3
1.3	Naming Conventions	4
1.4	Data Types	4
1.5	Windows Calling Convention	5
1.6	Near and Far Pointers	6

1

.

2

.

3

1.1 Introduction

This manual describes the functions, data types, data structures, files, and assembly-language macros used by Window applications to access the Windows system.

1.2 Notational Conventions

The following notational conventions are observed throughout this manual:

Convention	Meaning
boldface	Boldface is used for keywords, such as function, data type, structure, and macro names. These names are spelled exactly as they should appear in source programs. For example, in ReleaseCapture() ReleaseCapture is bold to indicate that it is the name of a function.
<i>italics</i>	Italics are used for placeholders, i.e., descriptive names that represent parameters or variables that the programmer must supply. For example, in ReplyMessage(lReply) <i>lReply</i> is a parameter passed to the ReplyMessage function.
(Parentheses)	Parentheses enclose the list of parameters to be passed to a function. For example, in ScreenToClient(hWnd, lpPoint) <i>hWnd</i> and <i>lpPoint</i> are parameters to be passed to the ScreenToClient function.
:	The colon separates a function's parameter list from the return value. Return values always appear after the colon. For example, GetMenu(hWnd):hMenu indicates that the value of <i>hMenu</i> is to be returned when GetMenu is called.

1.3 Naming Conventions

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

RegisterClass
CreateWindow
SetMapMode

1.4 Data Types

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

Prefix	Meaning
c	character (a one-byte value)
b	Boolean (a nonzero value means true, zero means false)
f	bit flags packed into a 16-bit integer
n	short (16-bit) integer
l	long (32-bit) integer
w	short (16-bit) unsigned integer
dw	long (32-bit) unsigned integer
h	16-bit handle

p	short (16-bit) pointer
lp	long (32-bit) pointer
pt	for X,Y coordinates packed into an unsigned 32-bit integer
rgb	for an rgb color value packed into 32-bit integer

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

Some examples of parameter and variable names are shown below:

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

1.5 Windows Calling Convention

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

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

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

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

1.6 Near and Far Pointers

A "near" pointer is a short pointer (16 bits) that consists of just an offset from one of the current segment register values. A "far" pointer is a long pointer (32 bits) that consists of a full segmented address: the segment address is in the high-order word, and the offset is in the low-order word. The default size of pointers in your program depends on the language you are using and the memory model.

In some cases Windows explicitly requires a far pointer. In particular, whenever you pass Windows a pointer to a function that will later be called back by Windows, the pointer must be a far pointer. (The reference manual tells you when a far pointer is required.)

To obtain a far pointer when the default size of your pointers is near, you must explicitly declare the pointer to be a far pointer. See your language documentation for details on declaring far pointers.

Chapter 2

Window Functions

2.1	Introduction	9
2.2	Main Function	9
2.3	Message Functions	10
2.4	Window Function	20
2.5	Default Window Function	21
2.6	Window Class Functions	22
2.7	Window Creation Functions	26
2.8	Window Display and Movement Functions	32
2.9	Dialog Box Functions	39
2.10	Clipboard Functions	47
2.11	Input Functions	55
2.12	Menu Functions	61
2.13	Window Painting Functions	70
2.14	Scrolling Functions	77
2.15	Property List Functions	81
2.16	Window Attribute Functions	84
2.17	Error Functions	87
2.18	Cursor Functions	90
2.19	Caret Functions	92
2.20	Coordinate Functions	95
2.21	Rectangle Functions	97
2.22	System Information Functions	100
2.23	Window Hook Function	103

1

2

3

2.1 Introduction

This chapter describes how to create and control windows in Windows applications. An application controls its own execution by reading messages from the application queue and dispatching the message to an appropriate window function. Messages not handled by the application are sent to a default window function supplied by Windows.

2.2 Main Function

The main function is an application-supplied function that serves as the entry point for program execution. Usually, the **WinMain** initializes the application, then controls the application's execution by processing messages from the Windows application queue.

Every application must have a main function of the form and function described below.

WinMain (*hInstance*, *hPrevInstance*, *lpCmdLine*, *nCmdShow*) : *nExitCode*

Purpose This function is the entry point for execution of a Windows application. Typically, the **WinMain** function initializes the application's data, registers the application's window class, creates and displays a window, then enters a loop to process messages from the Windows application queue.

The loop continues until a WM_QUIT message is received (either through calling **PostQuitMessage** or through receiving a nonzero return value from **GetMessage**.) When this occurs, the **WinMain** function must exit, passing the *wParam* parameter of the message to Windows as the exit value.

Parameters *hInstance* is the instance handle of the new instance.

hPrevInstance is the module instance handle of the previous module instance. It is NULL if this is the first instance.

lpCmdLine is a long pointer to a null-terminated command line.

nCmdShow is a short integer value specifying whether or not the window should be initially displayed, left iconic, or hidden. The parameter must be passed to the **ShowWindow** function after the application's window has been created.

Return Value *nExitCode* is the *wParam* value from the last message received (the message to quit the window).

Notes The **WinMain** function must use the Pascal calling convention.

WinMain does not have to be exported.

2.3 Message Functions

Applications use the message functions to read and process the messages in the application queue.

PostQuitMessage (*nExitCode*)

Purpose This function informs Windows that the application wishes to terminate execution. It is typically used in response to a WM_DESTROY message.

PostQuitMessage posts a WM_QUIT message to the application and returns immediately; the function merely informs the system that the application wants to quit sometime in the future.

When the application receives the WM_QUIT message, it should exit the message loop in the main function and return control to Windows. The exit code returned to Windows must be the *nParam* parameter of the WM_QUIT message.

Parameter *nExitCode* is the *wParam* parameter of the WM_QUIT message.

Return Value None.

GetMessage (*lpMsg*, *hWnd*, *wMsgFilterMin*, *wMsgFilterMax*) : *bContinue*

Purpose This function retrieves a message from the application queue and places the message in the data structure pointed to by *lpMsg*. If no message is available, **GetMessage** yields control to other applications until a message becomes available.

GetMessage retrieves only messages associated with the window specified by *hWnd* and within the range of message values given by *wMsgFilterMin* and *wMsgFilterMax*. If *hWnd* is NULL, **GetMessage** retrieves messages for any window belonging to the application making the call. (**GetMessage**

does not retrieve messages for windows belonging to other applications.) If *wMsgFilterMin* and *wMsgFilterMax* are both 0, **GetMessage** returns all available messages (no filtering is performed).

The constants WM_KEYFIRST and WM_KEYLAST can be used as filter values to retrieve all messages related to keyboard input; WM_MOUSEFIRST and WM_MOUSELAST can be used to retrieve all mouse-related messages.

Parameters *lpMsg* is a long pointer to a **MSG** data structure.

hWnd is a handle to the window whose messages are to be examined. If *hWnd* is NULL, **GetMessage** retrieves messages for any window belonging to the application making the call.

wMsgFilterMin is an unsigned short integer value specifying the lowest message to be examined.

wMsgFilterMax is an unsigned short integer value specifying the highest message to be examined.

Return Value *bContinue*, a Boolean value, is nonzero if a message other than WM_QUIT is retrieved. It is zero if the WM_QUIT message is retrieved.

The return value is usually used to decide whether to terminate the application's main loop and exit the program.

Notes In addition to yielding control to other applications when no messages are available, **GetMessage** and **PeekMessage** also yield control when WM_PAINT or WM_TIMER messages are available.

The **GetMessage**, **PeekMessage**, and **WaitMessage** functions are the only way to let other applications run. If your application does not call any of these functions for long periods of time, other applications do not get a chance to run.

When **GetMessage**, **PeekMessage**, and **WaitMessage** yield control to other applications, the stack and data segments of the application calling the function may move in memory to accommodate the changing memory requirements of other applications. If the application has stored

long pointers to objects in the data or stack segment (i.e., global or local variables), these pointers can become invalid after a call to **GetMessage**, **PeekMessage**, or **WaitMessage**. However, the *lpMsg* parameter of the called function remains valid in any case.

PeekMessage (*lpMsg*, *hWnd*, *wMsgFilterMin*, *wMsgFilterMax*,
bRemoveMsg) : *bPresent*

Purpose This function checks the application queue for a message and places the message (if any) in the data structure pointed to by *lpMsg*. **PeekMessage** returns immediately, whether a message is present or not.

PeekMessage retrieves only messages associated with the window specified by *hWnd* and within the range of message values given by *wMsgFilterMin* and *wMsgFilterMax*. If *hWnd* is NULL and *wMsgFilterMin* and *wMsgFilterMax* are both 0, **PeekMessage** checks the entire queue for messages.

The constants WM_KEYFIRST and WM_KEYLAST can be used as filter values to get all key messages; WM_MOUSEFIRST and WM_MOUSELAST can be used to get all mouse messages.

Parameters *lpMsg* is a long pointer to a data structure having **MSG** type.

hWnd is a handle to the window whose messages are to be examined.

wMsgFilterMin is an unsigned short integer value specifying the lowest message to be examined.

wMsgFilterMax is an unsigned short integer value specifying the highest position to be examined.

bRemoveMsg is a Boolean value that determines whether or not the message should be removed from the application queue. If *bRemoveMsg* is nonzero, the message is removed. If zero, it remains in the queue and can be read by the next call to **GetMessage** or **PeekMessage**.

Return Value *bPresent*, a Boolean value, is nonzero if a message is available. Otherwise, it is zero.

Notes WM_PAINT messages cannot be removed from the queue using **PeekMessage**. They remain in the queue until processed.

The **GetMessage**, **PeekMessage**, and **WaitMessage** functions yield control to other applications. These calls are the only way to let other applications run. If your application does not call any of these functions for long periods of time, other applications do not get a chance to run.

When **GetMessage**, **PeekMessage**, and **WaitMessage** yield control to other applications, the stack and data segments of the application calling the function may move in memory to accommodate the changing memory requirements of other applications. If the application has stored long pointers to objects in the data or stack segment (i.e., global or local variables), these pointers can become invalid after a call to **GetMessage**, **PeekMessage**, or **WaitMessage**. However, the *lpMsg* parameter of the called function remains valid in any case.

WaitMessage ()

<i>Purpose</i>	This function is used to yield control to other applications when an application has no other tasks to perform. WaitMessage suspends the application and does not return until a new message is placed in the application's queue.
<i>Parameters</i>	None.
<i>Return Value</i>	None.
<i>Notes</i>	The construction

```
if (!PeekMessage(...))
    WaitMessage();
```

is equivalent to a call to **GetMessage**.

The **GetMessage**, **PeekMessage**, and **WaitMessage** functions yield control to other applications. These calls are the only way to let other applications run. If your application does not call any of these functions for long periods of time, other applications do not get a chance to run.

When **GetMessage**, **PeekMessage**, and **WaitMessage** yield control to other applications, the stack and data segments of the application calling the function may move in

memory to accommodate the changing memory requirements of other applications. If the application has stored long pointers to objects in the data or stack segment (i.e., global or local variables), these pointers can become invalid after a call to **GetMessage**, **PeekMessage**, or **WaitMessage**.

GetMessagePos () : *ptPos*

Purpose This function returns a long value representing the mouse position, in screen coordinates, when the last message obtained by **GetMessage** occurred.

Parameters None.

Return Value *ptPos* is an unsigned long value containing the x and y coordinates of the mouse position. The x coordinate is in the low-order word and the y coordinate is in the high-order word. If the return value is assigned to a variable, the **MAKEPOINT** macro can be used to obtain a **POINT** structure from *ptPos*; the **LOWORD** or **HIGHWORD** macro can be used to extract the x or the y coordinate.

Notes To obtain the current position of the mouse cursor instead of the position when the last message occurred, use the **GetCursorPos** function.

GetMessageTime () : *lTime*

Purpose This function returns the message time for the last message retrieved by **GetMessage**. The message time is a long integer specifying the elapsed time (in milliseconds) since the system was booted to the time the message was created (placed in the application queue).

Parameters None.

Return Value *lTime* is a long integer value specifying the message time.

Notes Do not assume that the *lTime* value is always increasing. If the system has been on a long time, the *lTime* value may be "wrapped around" to start again at zero. To calculate time delays between messages, subtract the time of the second message from the time of the first message.

GetCurrentTime () : lTime

Purpose This function returns the time elapsed since the system was booted to the current time.

Parameters None.

Return Value *lTime* is a long integer specifying the elapsed time in milliseconds.

Notes The value returned by **GetCurrentTime** is in the same units as the value returned by **GetMessageTime**. For instance, these two values can be subtracted to determine the time elapsed since a message occurred.

TranslateMessage (*lpMsg*) : *bTranslated*

Purpose This function translates virtual keystroke messages into character messages, as follows:

- WM_KEYDOWN and WM_KEYUP messages are translated into WM_CHAR and WM_DEADCHAR messages.
- WM_SYSKEYDOWN and WM_SYSKEYUP are translated into WM_SYSCHAR and WM_SYSDEADCHAR messages.

The character messages are posted to the application queue, to be read the next time the application calls **GetMessage** or **PeekMessage**.

Parameters *lpMsg* is a long pointer to a data structure with **MSG** type, retrieved through **GetMessage** or **PeekMessage**.

Return Value *bTranslated*, a Boolean value, is nonzero if the message was translated (i.e., character messages were posted to the application queue). Otherwise, *bTranslated* is zero.

Notes **TranslateMessage** does not modify the message given by *lpMsg*.

An application should not call **TranslateMessage** if the application processes virtual key messages for some other purpose. For instance, an application should not call **TranslateMessage** if **TranslateAccelerator** returns nonzero.

TranslateAccelerator (*hWnd*, *hAccTable*, *lpMsg*) : *bTranslated*

Purpose

This function processes keyboard accelerators for menu commands. It translates WM_KEYUP and WM_KEYDOWN messages to WM_COMMAND or WM_SYSCOMMAND messages, if there is an entry for the key in the application's accelerator table. The high-order word of the *lParam* parameter of the WM_SYSCOMMAND or WM_COMMAND message contains the value 1 to differentiate the message from messages sent by menus or controls.

The WM_COMMAND or WM_SYSCOMMAND messages are sent directly to the window, rather than being posted to the application queue. **TranslateAccelerator** does not return until the command is processed.

Accelerator keystrokes that are defined to select items from the system menu are translated into WM_SYSCOMMAND messages; all other accelerators are translated into WM_COMMAND messages.

Parameters

hWnd is a handle to the window whose messages are to be translated.

hAccTable is a handle to an accelerator table (loaded using **LoadAccelerators**).

lpMsg is a long pointer to a message retrieved using **GetMessage** or **PeekMessage**. The message must be a data structure having MSG type.

Return Value

bTranslated, a Boolean value, is nonzero if translation occurred, zero otherwise.

Notes

When **TranslateAccelerator** returns nonzero (meaning that the message was translated), the application should NOT process the message again using **TranslateMessage**.

Commands in accelerator tables do not have to correspond to menu items.

If the accelerator command does correspond to a menu item, the application is sent WM_INITMENU and WM_INITMENUPOPUP messages, just as if the user were trying to pull down the menu. However, these messages are not sent if any of the following conditions hold:

- The window is disabled.
- The menu item is disabled.
- The command is not in the system menu and the window is iconic.
- A mouse capture is in effect (see **SetCapture**).

If the window is the active window and there is no keyboard focus (generally true if the window is iconic), then WM_SYSKEYUP and WM_SYSKEYDOWN messages are translated instead of WM_KEYUP and WM_KEYDOWN messages.

If an accelerator keystroke corresponding to a menu item occurs when the window owning the menu is iconic, no WM_COMMAND message is sent. However, if an accelerator keystroke occurs that does not match any of the items on the window's menu or on the system menu, a WM_COMMAND message is sent, even if the window is iconic.

DispatchMessage (*lpMsg*) : *lResult*

Purpose This function passes the message in the **MSG** structure pointed to by *lpMsg* to the window function of the specified window.

Parameters *lpMsg* is a long pointer to a data structure having **MSG** type. The structure must contain valid message values.

If *lpMsg* points to a WM_TIMER message and the *lParam* parameter of the WM_TIMER message is not NULL, then the *lParam* parameter is the address of a function which is called instead of the window function.

Return Value *lReply*, a long value, is the value returned by the window function. Its meaning depends on the message being dispatched, but generally *lReply* is ignored.

SendMessage (*hWnd*, *wMsg*, *wParam*, *lParam*) : *lReply*

Purpose This function sends a message to a window or windows. **SendMessage** does not return until the message has been processed. If the window receiving the message is part of the same application, the window function is called immediately, as a subroutine.

If the window is part of another task, Windows switches to the appropriate task and calls the appropriate window function, passing the message to the window function. The message is not placed in the destination application's queue.

Parameters *hWnd* is the handle of the window to receive the message. If the *hWnd* parameter is FFFF (hexadecimal), the message is sent to all tiled or popup windows in the system. (The message is not sent to child windows.)

wMsg is an unsigned short integer specifying the message to be set.

wParam is an unsigned short integer specifying additional message information.

lParam is a 32-bit integer specifying additional message information.

Return Value *lReply*, a long integer, is the value returned by the window function that received the message. Its value depends on the message being sent.

PostMessage (*hWnd*, *wMsg*, *wParam*, *lParam*) : *bPosted*

Purpose This function places a message in a window's application queue and returns without waiting for the corresponding window to process the message. The posted message can be retrieved by calls to **GetMessage** or **PeekMessage**.

Parameters *hWnd* is the handle of the window to receive the message. If the *hWnd* parameter is FFFF (hexadecimal), the message is sent to all tiled or popup windows in the system. (The message is not sent to child windows.)

wMsg is an unsigned short integer specifying the type of message posted.

wParam is a short integer value specifying additional message information.

lParam is a long integer value specifying additional message information.

Return Value *bPosted*, a Boolean value, is nonzero if the message is posted. Otherwise, it is zero.

ReplyMessage (*lReply*)

Purpose This function is used to reply to a message sent through **SendMessage** without returning control to the function calling **SendMessage**. Ordinarily, **SendMessage** waits for the return value of the function to which the message was sent. **ReplyMessage** allows the function receiving the message to send back a value without returning.

ReplyMessage has no effect if the message was not sent through **SendMessage**.

Parameters *lReply* is a long integer specifying the result of the message processing. The possible values depend on the actual message sent.

Return Value None.

PostAppMessage (*hTask*, *wMsg*, *wParam*, *lParam*) : *bPosted*

Purpose This function posts a message to an application identified by a task handle and returns without waiting for the application to process the message. The application receiving the message obtains the message by calling **GetMessage** or **PeekMessage**. The *hWnd* field of the returned **MSG** structure is NULL.

Parameters *hTask* is the task handle of the application to receive the message.

wMsg is an unsigned short integer specifying the type of message posted.

wParam is an unsigned short integer specifying additional message information.

lParam is a long integer specifying additional message information.

Return Value *bPosted*, a Boolean value, is nonzero if the message is posted. Otherwise, it is zero.

RegisterWindowMessage (*lpString*) : *wMsg*

Purpose This function defines a new window message that is guaranteed to be unique throughout the system. The returned message value can be used when calling **SendMessage** or **PostMessage**.

RegisterWindowMessage is typically used for communication between two cooperating applications.

If the same message string is registered by two different applications, the same message value is returned. The message remains registered until the user ends the Windows session.

<i>Parameter</i>	<i>lpString</i> is a long pointer to the message string to be registered.
<i>Return Value</i>	<i>wMsg</i> is an unsigned short integer in the range C000 to FFFF (hexadecimal) if the message is successfully registered. <i>wMsg</i> is zero if an error occurs.
<i>Notes</i>	RegisterWindowMessage need only be used when the same message must be understood by more than one application. For sending private messages within an application, an application can use any integer constant in the range WM_USER to BFFF (hexadecimal).

2.4 Window Function

The window function is an application-supplied function that processes the formatted messages sent to it by Windows or the application's main function. The window function can have any name (it is called **WndProc** below), but its format and function must be as follows.

WndProc (*hWnd*, *wMsg*, *wParam*, *lParam*) : *lReply*

Purpose This routine processes messages sent to it by Windows or the application's main function.

Parameters *hWnd* is the handle of the window receiving the message.

wMsg is an unsigned short integer containing the message number.

wParam is an unsigned short integer containing additional message-dependent information.

lParam is a long integer specifying additional message-dependent information.

Return Value *lReply* is a long integer specifying the result of the message processing. The possible return values depend on the actual message sent.

Note This function must use the Pascal calling conventions, must return a long value, and must be declared FAR.

The *lParam* parameter often contains two 16-bit values instead of a single 32-bit value. You can use the **HIBYTE** and **LOWORD** macros to extract these values. You can also use the **HIBYTE** and **LOBYTE** macros to extract the high- and low-order bytes of a given word.

2.5 Default Window Function

The default window function (supplied by Windows) carries out default processing of messages. The application-supplied window function (described in the previous section) must call the default window function to process any messages that it does not process itself.

DefWindowProc (*hWnd*, *wMsg*, *wParam*, *lParam*) : *lReply*

Purpose This function provides default processing for any Windows messages that a given application chooses not to process.

Parameters *hWnd* is the handle of the window passing the message.

wMsg is an unsigned short integer specifying the message number.

wParam is an unsigned short integer specifying additional message-dependent information.

lParam is a long integer specifying additional message-dependent information.

Return Value *lReply* is a long integer specifying the result of the message processing. The possible return values depend on the actual message sent.

2.6 Window Class Functions

The **RegisterClass** function defines and registers window classes to be used by an application. Every window created by an application must belong to a window class. The window class defines the class attributes of the window.

Any number of window classes can be registered. Once a class has been registered, Windows permits the application to create any number of windows belonging to that class.

The rest of the functions described in this section retrieve information about a window class and set window class attributes.

RegisterClass (*lpWndClass*) : *bRegistered*

Purpose This function registers a window class for subsequent use in calls to the **CreateWindow** function. The window class has the class attributes defined by the contents of the data structure pointed to by *lpWndClass*. The windows created with the registered class have a common appearance and function.

Parameters *lpWndClass* is a long pointer to a data structure having **WNDCLASS** type. The structure must be filled with the appropriate class attributes before being passed to the function.

Return Value *bRegistered*, a Boolean value, is nonzero if the class is registered. Otherwise, it is zero.

Notes If two classes having the same name are registered, the most recently registered class is recognized; the other is ignored.

To avoid conflicts with the class names of other applications, it is a good idea to use a derivation of the application's module name (from the .DEF file) for the window class name.

GetClassName (*hWnd*, *lpClassName*, *nMaxCount*) : *nCopied*

Purpose This function retrieves the class name of the window specified by *hWnd*.

Parameters *hWnd* is a handle to the window whose class name is to be retrieved.

lpClassName is a long pointer to a buffer to receive the class name.

nMaxCount is a short integer specifying the maximum number of bytes to be stored in *lpClassName*.

Return Value *nCopied*, a short integer value, is the number of characters actually copied to *lpClassName*. The return value is zero if the specified class is not a valid class.

GetClassWord (*hWnd*, *nIndex*) : word

Purpose This function retrieves the word at the given *nIndex* in the **WNDCLASS** structure of the window specified by *hWnd*.

Parameter *hWnd* is a handle to a window.

nIndex must be one of the following short integer values, specifying which word of the structure is to be retrieved.

- GCW_HBRBACKGROUND
- GCW_HCURSOR
- GCW_HICON
- GCW_HINSTANCE
- GCW_CBWNDEXTRA
- GCW_CBCLSEXTRA
- GCW_STYLE

Return Value *word* is the value retrieved from the **WNDCLASS** structure.

Note To access any extra bytes that were allocated when the window class structure was created, use positive offsets as indexes, starting at zero for the first byte of the extra space.

GetClassLong (*hWnd*, *nIndex*) : long

Purpose This function retrieves the long value at the given *nIndex* in the **WNDCLASS** structure of the window specified by *hWnd*.

Parameter *hWnd* is a handle to a window.

nIndex must be one of the following short integer values, specifying which long value of the structure is to be retrieved.

- GCL_MENUNAME
- GCL_WNDPROC

Return Value *long* is the long value retrieved from the **WNDCLASS** structure.

Notes To access any extra bytes that were allocated when the window class structure was created, use positive offsets as indexes, starting at zero for the first byte of the extra space.

SetClassWord (*hWnd*, *nIndex*, *wNewWord*) : *wOldWord*

Purpose This function replaces the word at the given *nIndex* in the **WNDCLASS** structure of the window specified by *hWnd*.

Parameter *hWnd* is a handle to a window.

nIndex must be one of the following short integer values, specifying which word of the structure is to be changed.

GCW_HBRBACKGROUND
GCW_HCURSOR
GCW_HICON
GCW_HINSTANCE
GCW_CBWNDEXTRA
GCW_CBCLSEXTRA
GCW_STYLE

wNewWord is the replacement value.

Return Value *wOldWord* is the old value of the specified word.

Note The **SetClassWord** function should be used with great care. For example, it is possible to change the background color for a class using **SetClassWord**, but this change does not cause all windows belonging to the class to be repainted immediately.

SetClassLong (*hWnd*, *nIndex*, *lNewLong*) : *lOldLong*

Purpose This function replaces the long value at the given *nIndex* in the **WNDCLASS** structure of the window specified by *hWnd*.

Parameter *hWnd* is a handle to a window.

nIndex must be one of the following short integer values, specifying which word of the structure is to be changed.

GCL_MENUNAME
GCL_WNDPROC

lNewLong is the replacement value.

Return Value *lOldLong* is the old value of the specified long integer.

Notes Any function set using **SetClassLong** with the GCL_WNDPROC index should be exported in the module definition file.

CallWindowProc (*lpPrevWndFunc*, *hWnd*, *wMsg*, *wParam*, *lParam*)
: *lReply*

Purpose This function passes message information to the function specified by *lpPrevWndFunc*. **CallWindowProc** is used for window “subclassing.” Normally, all windows with the same class share the same window function. A subclass is a window or set of windows belonging to the same window class whose messages are intercepted and processed by another function (or functions) before being passed to the class window function.

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

Parameters *lpPrevWndFunc* is a long pointer to the previous window function.

hWnd is the window handle of the window passing the message.

wMsg is an unsigned integer specifying the message number.

wParam is an unsigned integer specifying additional message-dependent information.

lParam is an unsigned integer specifying additional message-dependent information.

Return Value *lReply* is a long integer specifying the result of the message processing. The possible return values depend on the actual message sent.

2.7 Window Creation Functions

This section describes the functions used to create, destroy, modify, and obtain information about tiled, popup, and child windows. An application creates a window by providing all the data needed by Windows to draw the image of the window on the display screen. When the application no longer needs the window, the window is destroyed and its data are removed from memory.

CreateWindow (*lpClassName*, *lpWindowName*, *dwStyle*, *X*, *nWidth*,
nHeight, *hWndParent*, *hMenu*, *hInstance*, *lpParam*) : *hWnd*

Purpose This function creates tiled, popup, and child windows.

Parameters *lpClassName* is a long pointer to a null-terminated ASCII string naming the window class.

lpWindowName is a long pointer to a null-terminated ASCII string representing the window name. For windows with caption bars, the window name is displayed as the caption. When using **CreateWindow** to create controls such as buttons, check boxes, and text controls, *lpWindowName* specifies the text of the control.

dwStyle is a long unsigned integer specifying the style of window being created. It can be any combination of the styles given in Table 2.1. The control styles given in Chapter 6, Table 6.2, can also be used. Styles can be combined using the bitwise OR operator.

X and *Y* are short integer values specifying the initial position of the window. The meaning depends on the window style:

Style	Meaning
Tiled windows	<i>X</i> is ignored. If the WS_VISIBLE style is specified, then the <i>Y</i> parameter is assumed to be a ShowWindow command (SHOW_ICONWINDOW, SHOW_OPENWINDOW, SHOW_FULLSCREEN, or SHOW_OPENNOACTIVATE). This parameter can be the same as the <i>nCmdShow</i> parameter passed to the WinMain function.

Popup windows *X* and *Y* specify (in screen coordinates) the location of the upper left corner of the popup window. For list boxes only, *X* and *Y* specify the location of the upper left corner of the window's client area.

Child windows *X* and *Y* specify the location of the upper left corner of the child window in the client coordinates of the parent window (relative to the top left corner of the parent's client area). For list boxes only, *X* and *Y* specify the location of the upper left corner of the window's client area.

nWidth and *nHeight* are short integer values specifying the size of the window. The meaning depends on the window style:

Style	Meaning
Tiled windows	<i>nHeight</i> and <i>nWidth</i> are ignored for tiled windows. To reflect this fact, it is recommended that these parameters be set to zero.
Popup windows	<i>nWidth</i> and <i>nHeight</i> specify (in device units) the width and height of the popup window. For list boxes only, <i>nWidth</i> and <i>nHeight</i> specify (in device units) the width and height of the window's client area.
Child windows	<i>nWidth</i> and <i>nHeight</i> specify (in device units) the width and height of the child window. For list boxes only, <i>nWidth</i> and <i>nHeight</i> specify (in device units) the width and height of the window's client area.

hWndParent is a handle to a parent window. *hWndParent* must be NULL when creating a tiled window. When creating a child window, a valid value for *hWndParent* must be given. When creating a popup window, *hWndParent* can be non-null, but does not have to be.

hMenu is a handle to the menu to be used with the window.

If *hMenu* is NULL, the class menu is used. When creating a child window, *hMenu* specifies the child window ID, a short integer value. The child window ID is determined by the application and should be unique among child windows with the same parent.

hInstance is the instance handle associated with the window (passed to the application in the call to **WinMain**).

lpParam is a long pointer to a long value that is passed to the application through the *lParam* parameter of the WM_CREATE message.

Return Value *hWnd* is a handle to the new window. It is NULL if the window is not created.

Table 2.1
Window Styles

Style	Meaning
WS_TILED	Create a tiled window.
WS_POPUP	Create a popup window. Cannot be used with WS_CHILD.
WS_CHILD	Create a child window. Cannot be used with WS_POPUP.
WS_ICONIC	Create a window that is initially iconic. For use with WS_TILED only.
WS_BORDER	Create a window that has a border.
WS_CAPTION	Create a window that has a caption bar (implies WS_BORDER).
WS_DLGFREAME	Create a window with a double border but no caption.
WS_SYSMENU	Create a window that has a system menu box in its caption bar. Used only for windows with caption bars. If used with a child window, creates a close box instead of a system menu box.
WS_SIZEBOX	Create a window that has a size box. Used only for windows with a caption bar or with vertical and horizontal scroll bars.
WS_VSCROLL	Create a window that has a vertical scroll bar.
WS_HSCROLL	Create a window that has a horizontal scroll bar.

Table 2.1 (continued)

Style	Meaning
WS_CLIPCHILDREN	Exclude the area occupied by child windows when drawing within the parent window. Used when creating the parent window.
WS_CLIPSIBLINGS	Clip child windows relative to each other; that is, when a particular child window receives a paint message, clip all other overlapping child windows out of the region of the child window to be updated. (If WS_CLIPSIBLINGS is not given and child windows overlap, it is possible, when drawing in the client area of a child window, to draw in the client area of a neighboring child window.) For use with WS_CHILD only.
WS_VISIBLE	Create a window that is initially visible.
WS_DISABLED	Create a window that is initially disabled.
WS_TILEDWINDOW	Create a tiled window having the styles WS_TILED, WS_CAPTION, WS_SYSMENU and WS_SIZEBOX.
WS_POPUPWINDOW	Create a popup window that has the styles WS_POPUP, WS_BORDER, and WS_SYSMENU.
WS_CHILDWINDOW	Create a child window that has the style WS_CHILD.

IsWindow (hWnd) : bExists

Purpose This function determines whether the window identified by *hWnd* is a valid, existing window.

Parameters *hWnd* is a handle to a window.

Return Value *bExists*, a Boolean value, is nonzero if *hWnd* is a valid window handle. Otherwise, it is zero.

DestroyWindow (*hWnd*) : *bDestroyed*

Purpose This function sends a WM_DESTROY message to the window specified by *hWnd* and frees any memory that it occupied. If the window specified by *hWnd* is the parent of any windows (child style or popup style), these windows are automatically destroyed when the parent is destroyed.

Parameters *hWnd* is a handle to a window.

Return Value *bDestroyed*, a Boolean value, is nonzero if the window is destroyed. Otherwise, it is zero.

GetWindowWord (*hWnd*, *nIndex*) : *word*

Purpose This function retrieves information about the window identified by *hWnd*.

Parameter *hWnd* is a handle to a window.

nIndex must be one of the following integer values, specifying which word of information is to be retrieved.

Index	Meaning
GWW_HINSTANCE	Instance handle of the module owning the window
GWW_HWNDPARENT	Handle of the parent window, if any
GWW_HWNDTEXT	Handle to the window's caption
GWW_ID	Control ID of the window

Return Value *word* is the short integer value retrieved.

Notes To access any extra bytes that were allocated when the window class structure was created, use positive offsets as indexes, starting at zero for the first byte of the extra space.

GetWindowLong (*hWnd*, *nIndex*) : *long*

Purpose This function retrieves information about the window identified by *hWnd*.

Parameter *hWnd* is a handle to a window.

nIndex must be one of the following short integer values, specifying which long value of information is to be retrieved.

Index	Meaning
GWL_WNDPROC	Long pointer to window function
GWL_STYLE	Window style

Return Value *long* is the long integer value retrieved.

Notes To access any extra bytes that were allocated when the window class structure was created, use positive offsets as indexes, starting at zero for the first byte of the extra space.

SetWindowWord (*hWnd*, *nIndex*, *wNewWord*) : *wOldWord*

Purpose This function changes an attribute of the window specified by *hWnd*.

Parameter *hWnd* is a handle to a window.

nIndex specifies which word of information is to be changed.

Index	Meaning
GWW_HINSTANCE	Instance handle of the module owning the window
GWW_HWNDPARENT	Window handle of the parent window, if any
GWW_HWNDTEXT	Handle to the window's caption
GWW_ID	Control ID of the window

wNewWord is the replacement value.

Return Value *wOldWord* is the old value of the specified word.

Notes To access any extra bytes that were allocated when the window class structure was created, use positive offsets as indexes, starting at zero for the first byte of the extra space.

SetWindowLong (*hWnd*, *nIndex*, *lNewLong*) : *lOldLong*

Purpose This function changes an attribute of the window specified by *hWnd*.

Parameters *hWnd* is a handle to a window.

nIndex must be one of the following short integer values, specifying which long value of information is to be changed.

Index	Meaning
GWL_WNDPROC	Long pointer to window function
GWL_STYLE	Window style

lNewLong is the replacement value.

Return Value *lOldLong* is the old value of the specified long integer.

Notes To access any extra bytes that were allocated when the window class structure was created, use positive offsets as indexes, starting at zero for the first byte of the extra space.

Any function address set using **SetWindowLong** with the GWL_WNDPROC index should be created using **MakeProcInstance**.

2.8 Window Display and Movement Functions

This section describes functions used to show, hide, and move windows and to obtain information about the number and position of windows on the screen.

ShowWindow (*hWnd*, *nCmdShow*) : *bShown*

Purpose This function displays or removes the given window as specified by *nCmdShow*.

Parameters *hWnd* is a handle to a window.

nCmdShow is one of the following short integer values:

Value	Meaning
HIDE_WINDOW	Remove window from screen, but do not destroy it. Used for child style and popup style windows.
SHOW_OPENWINDOW	Display a previously hidden popup or child window, or display a tiled window for the first time.
SHOW_ICONWINDOW	Display a previously iconic window. Used for tiled windows only.
SHOW_FULLSCREEN	Use the full screen for displaying the window. Used for tiled windows only.
SHOW_OPENNOACTIVATE	Display the open window, but do not activate it. Used for tiled and popup windows.
<i>hWndSwap</i>	The window specified by <i>hWnd</i> is swapped with the window specified by <i>hWndSwap</i> , and the <i>hWndSwap</i> window is made iconic. Both <i>hWnd</i> and <i>hWndSwap</i> must specify tiled windows.
<i>IconSlot</i>	Make the window iconic and use the value given by <i>IconSlot</i> to determine the position of the icon (where 0 is the leftmost position on the icon bar). The nine high-order bits of <i>IconSlot</i> must be set; the seven low-order bits specify the icon position.

<i>Column</i>	Display the window in the column given by <i>Column</i> . The <i>Column</i> value must be the result of a bitwise OR operation with the desired column value and FF40 (hexadecimal). If <i>Column</i> is FF7F (FF40 and 3F are used in the OR operation), display the window as a new row at the bottom of the right-most column. If <i>Column</i> is FF7E (FF40 and 3E are used), display the window as a new column to the right of the last column. If <i>Column</i> is any other value, display the window as a new row at the bottom of the given column (where column 0 is the left-most column).
<i>Return Value</i>	<i>bShown</i> , a Boolean value, is nonzero if the window was previously visible. It is zero if the window was previously hidden.
<i>Notes</i>	ShowWindow must only be called once per program with the <i>nCmdShow</i> parameter from WinMain . Subsequent calls to ShowWindow must use one of the values listed above instead of the <i>nCmdShow</i> parameter.
OpenIcon (<i>hWnd</i>) : <i>bOpened</i>	
<i>Purpose</i>	This function opens the specified window by copying the window's caption and client area to the screen and removing its icon from the icon area. If other open windows already exist on the screen, OpenIcon directs Windows to resize and retile those windows to make room for the new window.
<i>Parameters</i>	<i>hWnd</i> is a handle to a window.
<i>Return Value</i>	<i>bOpened</i> , a Boolean value, is nonzero if the function is successful. Otherwise, it is zero.
<i>Notes</i>	This function has the same result as sending a WM_SYSCOMMAND message to an iconic window with the <i>wParam</i> parameter set to SC_ICON. Child style and popup style windows cannot be opened using this function.

CloseWindow (*hWnd*) : *bClosed*

- Purpose* This function closes the specified window. If the window is a tiled window, it is closed by removing the client area and caption of the open window from the display screen and moving the window's icon into the icon area of the screen. If the window is a popup window, it is hidden.
- Parameters* *hWnd* is a handle to the window to be closed.
- Return Value* *bClosed*, a Boolean value, is nonzero if the window is closed. Otherwise, it is 0.
- Notes* Child windows cannot be closed using this function. Any attempt to close a child window is ignored.
Calling this function has the same effect as a WM_SYSCOMMAND message with *wParam* set to SC_CLOSE.

MoveWindow (*hWnd*, *X*, *Y*, *nWidth*, *nHeight*, *bRepaint*)

- Purpose* This function causes a WM_SIZE message to be sent to the given window. The *X*, *Y*, *nWidth*, and *nHeight* parameters give the new size of the window.
- Parameters* *hWnd* is a handle to a popup style or child style window. *X* and *Y* are short integer values specifying the new coordinates of the upper left corner of the window. For popup style windows, *X* and *Y* are in screen coordinates (relative to the top left corner of the screen). For child style windows, they are in client coordinates (relative to the top left corner of the parent window's client area). *nWidth* and *nHeight* are short integer values specifying the new width and height of the window. *bRepaint*, a Boolean value, is nonzero if the window is to be repainted after moving. If *bRepaint* is zero, the window does not need repainting.
- Return Value* None.
- Notes* The WM_SIZE message created by this function gives the new width and height of the client area of the window, not the full window.

BringWindowToTop (*hWnd*)

Purpose This function brings a popup style or child style window to the top of a stack of overlapping windows. **BringWindowToTop** should be used to uncover any window that is partially or completely obscured by other, overlapping windows.

Parameters *hWnd* is a handle to a popup style or child style window.

Return Value None.

SetActiveWindow (*hWnd*) : *hWndPrev*

Purpose This function makes a tiled or popup style window the active window. If the window specified by *hWnd* does not currently have the input focus, the focus is set to NULL (input is ignored).

Parameters *hWnd* is a handle to the tiled or popup window to be made active.

Return Value *hWndPrev* is a handle to the window that was previously active.

Notes This function should not be called without good reason, since it allows an application to take over the active window and input focus arbitrarily. Normally, Windows takes care of all activation.

IsWindowVisible (*hWnd*) : *bVisible*

Purpose This function specifies whether or not the specified window is currently visible on the screen.

Parameters *hWnd* is a handle to a window.

Return Value *bVisible*, a Boolean value, is nonzero if the given window is visible. Otherwise, it is zero.

Notes **IsWindowVisible** returns nonzero even if the specified window is completely covered by another child style or popup style window.

AnyPopup () : bVisible

Purpose This function indicates whether or not a popup style window is visible on the screen. It searches the entire Windows screen, not just the caller's client area.

Parameters None.

Return Value *bVisible*, a Boolean value, is nonzero if a popup style window is visible. Otherwise, it is zero.

IsIconic (hWnd) : bIconic

Purpose This function specifies whether or not a window is open or closed (iconic).

Parameters *hWnd* is a handle to a window.

Return Value *bIconic*, a Boolean value, is nonzero if the window is closed (iconic). It is zero if the window is open.

EnumWindows (lpEnumFunc, lParam) : bDone

Purpose This function enumerates the windows on the screen by passing the handle of each window, in turn, to the application-supplied function pointed to by *lpEnumFunc*. The window handles are passed in the following order:

1. Tiled windows
2. Iconic windows
3. Popup style windows
4. Hidden popup style windows

Child style windows are not enumerated.

EnumWindows continues to enumerate windows until the called function returns zero or until the last window has been enumerated.

Parameters *lpEnumFunc* is a long pointer to the application-supplied enumeration function.

lParam is a long integer that is passed to the enumeration function for the application's use.

Return Value *bDone*, a Boolean value, is nonzero if all windows have been enumerated. Otherwise, it is zero.

Notes The address passed as the *lpEnumFunc* parameter must be created using **MakeProcInstance**.

The enumeration function must use the Pascal calling convention and must be declared FAR. Its form is as follows:

functionname(hWnd, lParam)

hWnd is the window handle and *lParam* is the long parameter passed to the **EnumWindows** function. The function returns a Boolean value: nonzero to continue enumeration, zero to terminate it.

EnumChildWindows (*hWndParent*, *lpEnumFunc*, *lParam*) : *bDone*

This function enumerates the child style windows belonging to the specified parent by passing the handle of each child style window, in turn, to the application-supplied function pointed to by *lpEnumFunc*.

EnumChildWindows continues to enumerate windows until the called function returns zero or until the last child style window has been enumerated.

Parameters *hWndParent* is a handle to the parent window whose child windows are to be enumerated.

lpEnumFunc is a long pointer to the application-supplied enumeration function.

lParam is a long integer that is passed to the enumeration function for the application's use.

Return Value *bDone*, a Boolean value, is nonzero if all child windows have been enumerated. Otherwise, it is zero.

Notes This function does not enumerate popup style windows belonging to *hwndParent*.

The address passed as the *lpEnumFunc* parameter must be created using **MakeProcInstance**.

The enumeration function must use the Pascal calling convention and must be declared FAR. Its form is as follows:

functionname(hWnd, lParam)

hWnd is the window handle and *lParam* is the long parameter passed to the **EnumChildWindows** function. The function returns a Boolean value: nonzero to continue enumeration, zero to terminate it.

2.9 Dialog Box Functions

This section describes the functions used to create, access, modify, and destroy dialog boxes. A dialog box is a window belonging to a predefined window class. Dialog boxes provide a convenient way to interact with the user temporarily when input through a menu is not sufficient.

**CreateDialog (*hInstance*, *lpTemplateName*, *hWndParent*, *lpDialogFunc*)
 : *hDlg***

Purpose This function creates a modeless dialog box. The dialog template given by *lpTemplateName* defines the attributes of the dialog box, such as the caption, menu, and controls. *hWndParent* identifies the application window that owns the dialog box. The dialog function pointed to by *lpDialogFunc* processes any messages received by the dialog box.

CreateDialog returns immediately after creating the dialog box. It does not wait for the dialog box to begin processing input.

Parameters *hInstance* is the instance handle of the module whose executable file contains the dialog template.

lpTemplateName is a long pointer to a character string naming the dialog template. The string must be a null-terminated ASCII string.

hWndParent is a handle to the window owning the dialog box.

lpDialogFunc is a long pointer to the dialog box function.

Return Value *hDlg* is a handle to the newly-created dialog box. It is NULL if the dialog box cannot be created.

Notes The address to be passed as the *lpDialogFunc* parameter must be created using **MakeProcInstance**.

The dialog function must use the Pascal calling convention and must be declared FAR. Its form is as follows:

functionname (hWnd, wMsg, wParam, lParam)

The dialog function returns a Boolean value that is nonzero if the function processes the message, zero if it doesn't. The parameters are identical to **WndProc** parameters. The *hWnd* parameter contains the window handle of the dialog box.

IsDialogMessage (*hDlg*, *lpMsg*) : *bUsed*

Purpose This function determines whether the given message is intended for the modeless dialog box specified by *hDlg*. If so, the message is processed and the function returns nonzero.

Parameters *hDlg* is a handle to a dialog box.

lpMsg is a long pointer to a MSG data structure.

Return Value *bUsed*, a Boolean value, is nonzero if the message has been processed, zero otherwise.

Notes If **IsDialogMessage** returns nonzero, the message has already been processed and should NOT be passed on to **TranslateMessage** or to **DispatchMessage**.

The **IsDialogMessage** can be used with any window containing controls (not just dialog box windows), if the application wants user actions in the window (such as tabbing or pressing the space bar) to have the same effect as in a modeless dialog box.

**DialogBox (*hInstance*, *lpTemplateName*, *hWndParent*, *lpDialogFunc*)
: *nResult***

Purpose This function creates a modal dialog box. The dialog template given by *lpTemplateName* defines the attributes of the dialog box, such as the caption, menu, and controls.

hWndParent identifies the application window that owns the dialog box. The dialog function pointed to by *lpDialogFunc* processes any messages received by the dialog box.

Parameters *hInstance* is the instance handle of the module whose executable file contains the dialog template.

lpTemplateName is a long pointer to a character string naming the dialog template. The string must be a null-terminated ASCII string.

hWndParent is a handle to the window owning the dialog box.

lpDialogFunc is a long pointer to the dialog box function.

Return Value *nResult* is the short integer value returned by the application's dialog function. *nResult* is -1 if insufficient memory is available to create the dialog box.

Notes

To create a second dialog box within the dialog function of another dialog box while the first dialog box still exists, pass the handle of the first dialog box into the second call to **DialogBox** as the parent window handle parameter. The first dialog box will be disabled while the second one receives input. Similarly, if a message box is created within the dialog function, pass the dialog box handle to the **MessageBox** function as the parent window handle parameter.

The address to be passed as the *lpDialogFunc* parameter must be created using **MakeProcInstance**.

The dialog function must use the Pascal calling convention and must be declared FAR. Its form is as follows:

functionname (hWnd, wParam, lParam)

The dialog function returns a Boolean value that is nonzero if the function processes the message, zero if it doesn't. The parameters are identical to **WndProc** parameters. The *hWnd* parameter contains the window handle of the dialog box.

EndDialog (*hDlg, nResult*)*Purpose*

This function frees resources and destroys windows associated with a modal dialog box. **EndDialog** is required to complete processing whenever the **DialogBox** function is used to create a dialog box.

Parameters

hDlg is a handle to a dialog box.

nResult, a short integer value, is the value returned by the initiating **DialogBox** call.

Return Value

None.

Notes

If necessary, **EndDialog** can be called during the processing of the WM_INITDIALOG message.

**DlgDirList (*hDlg, lpPathSpec, nIDListBox, nIDStaticPath, wFiletype*)
: bListed***Purpose*

This function creates a list that allows the user to select files or directories.

DlgDirList fills the list box specified by *nIDListBox* with the names of all files matching the path specification given

by *lpPathSpec*. Subdirectories are enclosed in square brackets; drives are shown in the form [-*x*], where *x* is the drive letter.

If *lpPathSpec* includes a drive and/or directory specification, the current drive and directory are changed to the designated drive and directory before the list box is filled. The text control identified by *nIDStaticPath* is also updated with the new drive and/or directory specification.

After the list box is filled, *lpPathSpec* is updated by removing the drive and/or directory portion of the path specification.

Parameters *hDlg* is the dialog box window handle.

lpPathSpec is a long pointer to a path specification string.

nIDListBox is the ID of a list box control. If *nIDListBox* is 0, **DlgDirList** assumes no list box exists and does not attempt to fill it.

nIDStaticPath is the ID of a static text control used for displaying the current drive and directory. If *nIDStaticPath* is 0, **DlgDirList** assumes that no such text control is present.

wFiletype is an unsigned integer whose bits specify MS-DOS® file attributes. *wFiletype* should be 0 for normal files (files with none of the attributes listed below), and should be set as shown below for other files.

Bit (hexadecimal)	Meaning
01	Read only file
02	Hidden file
04	System file
10	Subdirectory
20	Archive
4000	Drive bit
8000	Exclusive bit

If the “exclusive bit” is set, only files of the specified type are listed. Otherwise, files of the specified type are listed in addition to normal files.

Return Value *bListed* is nonzero if a listing was made, even an empty listing. A zero return value implies that the input string did not contain a valid search specification.

DlgDirSelect (*hDlg*, *lpString*, *nIDListBox*) : *bDirectory*

Purpose This function gets the current selection from a list box built by **DlgDirList** and copies the selection to the location specified by *lpString*.

If the current selection is a directory name or drive letter, **DlgDirSelect** removes the enclosing square brackets (and hyphens, for drive letters) so that the name or letter is ready to be inserted into a new path specification.

Parameters *hDlg* is a handle to a dialog box.

lpString is a long pointer to a string buffer to store the selection.

nIDListBox is the ID of a list box control in the dialog box.

Return Value *bDirectory*, a Boolean value, is nonzero if the current selection is a directory name. Otherwise, it is zero.

GetDlgItem (*hDlg*, *nIDDlgItem*) : *hCtl*

Purpose This function retrieves the handle of a dialog item (control) from the dialog box specified by *hDlg*.

Parameters *hDlg* is a handle to a dialog box.

nIDDlgItem is the integer *nID* of the item to be retrieved.

Return Value *hCtl* is a handle to the given dialog item. It is NULL if no item with the given *nIDDlgItem* exists.

SetDlgItemInt (*hDlg*, *nIDDlgItem*, *wValue*, *bSigned*)

Purpose This function sets the text of a dialog item (control) to the string representation of an integer value.

Parameters *hDlg* is a handle to a dialog box.

nIDDlgItem is an integer value identifying the dialog item to be modified.

wValue is the unsigned short integer value to be set.

bSigned specifies whether or not the integer value is signed. If *bSigned* is nonzero, *wValue* is a signed integer; if zero, *wValue* is unsigned.

Return Value None.

GetDlgItemInt (*hDlg*, *nIDDlgItem*, *lpTranslated*, *bSigned*) : *wValue*

Purpose This function translates the text of a dialog item into an integer value.

Parameters *hDlg* is a handle to the dialog box.

nIDDlgItem identifies the dialog item to be translated.

lpTranslated is a long pointer to a Boolean variable.

GetDlgItemInt places a nonzero value in the location pointed to by *lpTranslated* if the text is translated without errors. It places zero if any non-numeric characters occur in the text, or if the number represented by the text is too large to fit into an integer. If *lpTranslated* is NULL, the **GetDlgItemInt** does not warn about errors.

bSigned is a Boolean value specifying whether or not the value to be retrieved is signed. If *bSigned* is nonzero, **GetDlgItemInt** checks for a minus sign (-) before translating the number. Otherwise, **GetDlgItemInt** does not check for a minus sign.

Return Value *wValue* is an integer value specifying the translated value of the dialog item text. Since 0 is a valid return value, *lpTranslated* must be used to detect errors.

SetDlgItemText (*hDlg*, *nIDDlgItem*, *lpString*)

Purpose This function sets the caption or text of a dialog item (control) in the dialog box specified by *hDlg*.

Parameters *hDlg* is a handle to the dialog box.

nIDDlgItem is an integer value identifying the control item.

lpString is a long pointer to a null-terminated string representing the new caption or text.

Return Value None.

GetDlgItemText (*hDlg*, *nIDDlgItem*, *lpString*, *nMaxCount*) : *nCopied*

Purpose This function retrieves the caption or text associated with a dialog item (control) and copies it to the location pointed to by *lpString*.

Parameters *hDlg* is a handle to the dialog box.

nIDDlgItem is an integer value identifying the dialog item whose caption or text is to be retrieved.

lpString is a long pointer to the buffer to receive the text.

nMaxCount is a short integer value specifying the maximum length in bytes of the string to be copied to *lpString*. If the string is longer than *nMaxCount*, it is truncated.

Return Value *nCopied* is a short integer value specifying the number of characters copied to the buffer. *nCopied* is 0 if no text is copied.

CheckDlgButton (*hDlg*, *nIDButton*, *wCheck*)

Purpose This function places or removes a checkmark next to a button control, or changes the state of a 3-state button.

Parameters *hDlg* is a handle to the dialog box.

nIDButton is a short integer value specifying the button control to be modified.

wCheck is a short unsigned integer value specifying the action to take. If *wCheck* is nonzero, the **CheckDlgButton** places a checkmark next to the button; if zero, the checkmark (if any) is removed. For 3-state buttons, if *wCheck* is 2, the button is grayed; if *wCheck* is 1, it is checked; and if *wCheck* is zero, the checkmark (if any) is removed.

Return Value None.

IsDlgButtonChecked (*hDlg*, *nIDButton*) : *wCheck*

Purpose This function determines whether a button control has a checkmark next to it, or whether a 3-state button control is grayed, checked, or neither.

Parameters *hDlg* is a handle to the dialog box.

nIDButton is the integer ID of the button control.

Return Value *wCheck* is nonzero if the given control has a checkmark, zero if it does not. For 3-state buttons, *wCheck* is 2 if the button is grayed, 1 if the button has a checkmark next to it, and zero otherwise.

CheckRadioButton (*hDlg*, *nIDFirstButton*, *nIDLastButton*,
nIDCheckButton)

Purpose This function checks the radio button specified by *nIDCheckButton* and unchecks all other radio buttons in the group of buttons specified by *nIDFirstButton* and *nIDLastButton*.

Parameters *hDlg* is a handle to the dialog box.

nIDFirstButton and *nIDLastButton* are integer ID values for the first and last radio buttons in the group. The ID numbers of the radio buttons in the group must be sequential.

nIDCheckButton is an integer ID value identifying the radio button to be checked.

Return Value None.

SendDlgItemMessage (*hDlg*, *nIDDlgItem*, *wMsg*, *wParam*, *lParam*)
: *lResult*

Purpose This function sends a message to the dialog item (control) identified by *nIDDlgItem* within the dialog box specified by *hDlg*. **SendDlgItemMessage** does not return until the message has been processed.

Parameters *hDlg* is a handle to the dialog box.

nIDDlgItem is the integer ID value of the dialog item to receive the message.

wMsg is an unsigned short integer value representing a message.

wParam is a short integer value specifying additional message information.

lParam is a long integer value specifying additional message information.

Return Value *lResult*, a long integer, is equal to the value returned by the dialog item's window function.

MapDialogRect (*hDlg, lpRect*)

Purpose This function converts the dialog box coordinates given in *lpRect* to client coordinates.

Parameters *hDlg* is a handle to a dialog box.

lpRect is a long pointer to a data structure with **RECT** type that contains the dialog box coordinates to be converted. The dialog box coordinates in *lpRect* are replaced with the client coordinates.

Return Value None.

2.10 Clipboard Functions

This section describes the clipboard functions used to carry out data interchange between Windows applications. The clipboard provides a way for applications to pass data handles to other applications.

The clipboard recognizes five predefined data formats and allows applications to define additional formats. The predefined formats are text format, bitmap format, Microsoft Symbolic Link format (SYLK), Software Arts' Data Interchange Format (DIF), and metafile picture format.

OpenClipboard (*hWnd*) : *bOpened*

Purpose This function opens the clipboard for examination and prevents other applications from modifying the clipboard contents.

Parameters *hWnd* is a handle to the window to be associated with the open clipboard.

Return Value *bOpened*, a Boolean value, is nonzero if the clipboard is opened. If the clipboard has already been opened by another application, **OpenClipboard** returns zero.

CloseClipboard () : *bClosed*

Purpose This function closes the clipboard. **CloseClipboard** should be called when a window has finished examining or changing the clipboard. It lets other applications access the clipboard.

Parameters None.

Return Value *bClosed*, a Boolean value, is nonzero if the clipboard is closed. Otherwise, it is zero.

EmptyClipboard () : bEmptied

Purpose This function empties the clipboard and frees handles to data in the clipboard. It then assigns ownership of the clipboard to the window that currently has the clipboard open.

Parameters None.

Return Value *bEmptied*, a Boolean value, is nonzero if the clipboard is emptied. It is zero if an error occurs.

Note The clipboard must be open when **EmptyClipboard** is called.

GetClipboardOwner () : hWnd

Purpose This function retrieves the window handle of the current owner of the clipboard.

Parameters None.

Return Value *hWnd* is a handle to the window that owns the clipboard. It is NULL if the clipboard is not owned.

Note The clipboard can still contain data even if the clipboard is not currently owned.

SetClipboardData (wFormat, hMem) : hClipData

Purpose This function sets a data handle into the clipboard for the data specified by *hMem*. The data are assumed to have the format specified by *wFormat*.

After a clipboard data handle has been assigned, **SetClipboardData** frees the block identified by *hMem*.

Parameters *wFormat* is an unsigned short integer value that specifies a data format. It can be any one of the following predefined formats:

Format	Meaning
CF_TEXT	Text format. Each line ends with a carriage return/linefeed (CR-LF) combination. A NULL character signals the end of the data.
CF_BITMAP	Bitmap as defined by the BITMAP data structure.
CF_METAFILEPICT	Metafile picture as defined by the METAFILEPICT data structure.
CF_SYLK	Microsoft's Symbolic Link format.
CF_DIF	Software Arts' Data Interchange Format.
CF_OWNERDISPLAY	Owner display format. The clipboard owner is responsible for displaying and updating the clipboard application window, and will receive WM_PAINTCLIPBOARD, WM_SIZECLIPBOARD, WM_VSCROLLCLIPBOARD, WM_HSCROLLCLIPBOARD, and WMASKCBFORMATNAME messages. The <i>hMem</i> parameter must be NULL.
CF_DSPTEXT	Text display format associated with private format. <i>hMem</i> must be a handle to data that can be displayed in text format in lieu of the privately formatted data.
CF_DSPBITMAP	Bitmap display format associated with private format. <i>hMem</i> must be a handle to data that can be displayed in bitmap format in lieu of the privately formatted data.

CF_DSPMETAFILEPICT

Metafile picture display format associated with private format. *hMem* must be a handle to data that can be displayed in metafile picture format in lieu of the privately formatted data.

CF_PRIVATEFIRST

Range of integer values that can be used for private formats. Data handles associated with formats in this range will not be freed automatically; *hMem* must be freed by the application before the application terminates or when a WM_DESTROYCLIPBOARD message is received.

CF_GDIOBJFIRST

Range of integer values used for private formats where *hMem* is assumed to be a handle to a GDI object. Data handles associated with formats in this range are automatically freed by Windows (using DeleteObject) when the application terminates or the clipboard is emptied.

In addition to the above predefined formats, any format value registered through **RegisterClipboardFormat** can be used as the *wFormat* parameter.

hMem is a global handle to data in the specified format. *hMem* can be NULL; in this case, the application does not have to format the data and provide a handle to it until requested to do so through a WM_RENDERFORMAT message.

Return Value *hClipData* is the data handle assigned by the clipboard.

Note Once *hMem* has been passed to **SetClipboardData**, the block of data becomes the property of the clipboard. The application may read the data using *hClipData*, but should not free the block or leave it locked.

GetClipboardData (*wFormat*) : *hClipData*

Purpose This function retrieves data from the clipboard in the format given by *wFormat*. The clipboard must have been previously opened.

Parameters *wFormat* is an unsigned short integer value that specifies a data format. It can be any one of the following predefined formats:

Format	Meaning
CF_TEXT	Text format. Each line ends with a carriage return/linefeed (CR-LF) combination. A NULL character signals the end of the data.
CF_BITMAP	Bitmap as defined by the BITMAP data structure.
CF_METAFILEPICT	Metafile picture as defined by the METAFILEPICT data structure.
CF_SYLK	Microsoft's Symbolic Link format.
CF_DIF	Software Arts' Data Interchange Format.
CF_OWNERDISPLAY	Owner display format (can be used only by the clipboard owner).
CF_DSPTEXT	Text display format associated with private format.
CF_DSPBITMAP	Bitmap display format associated with private format.
CF_DSPMETAFILEPICT	Metafile picture display format associated with private format.

CF_PRIVATEFIRST Range of integer values that can be used for private formats. Data handles associated with formats in this range will not be freed automatically; *hMem* must be freed by the application before the application terminates or when a WM_DESTROYCLIPBOARD message is received.

CF_GDIOBJFIRST Range of integer values used for private formats where *hMem* is assumed to be a handle to a GDI object. Data handles associated with formats in this range are automatically freed by Windows (using **DeleteObject**) when the application terminates or the clipboard is emptied.

In addition to the above predefined formats, any format value registered through **RegisterClipboardFormat** can be used as the *wFormat* parameter.

Return Value *hClipData* is a handle to the data retrieved from the clipboard. It is NULL if there is an error.

Notes The available formats can be enumerated in advance by using **EnumClipboardFormats**.

The data handle returned by **GetClipboardData** is controlled by the clipboard, not the application. Thus, the application should copy the data immediately instead of relying on the data handle for long-term use. The application should not free the data handle or leave it locked.

RegisterClipboardFormat (*lpFormatName*) : *wFormat*

Purpose This function registers a new clipboard format whose name is pointed to by *lpFormatName*. The registered format can be used in subsequent clipboard functions as a valid format in which to render data, and it will appear in the clipboard's list of formats.

Parameters *lpFormatName* is a long pointer to a character string naming the new format. The string must be a null-terminated ASCII string.

Return Value *wFormat* is an unsigned short integer value specifying the newly registered format. If the identical format name has been registered before, even by a different application, the format's reference count is incremented and the same value is returned as when the format was originally registered. *wFormat* is 0 if the format cannot be registered.

Note The format value returned by **RegisterClipboardMessage** is in the range C000 to FFFF (hexadecimal).

CountClipboardFormats () : nCount

Purpose This function retrieves a count of the number of formats that the clipboard can render.

Parameters None.

Return Value *nCount* is an integer value specifying the number of data formats in the clipboard.

EnumClipboardFormats (wFormat) : nNextFormat

Purpose This function enumerates the formats found in a list of available formats belonging to the clipboard. On each call to **EnumClipboardFormats**, the *wFormat* parameter specifies a known available format, and the function returns the format that appears next in the list. The first format in the list can be retrieved by setting *wFormat* to 0.

Parameters *wFormat* is an unsigned short integer value specifying a known format.

Return Value *nNextFormat* is an integer value specifying the next known clipboard data format. It is 0 if *wFormat* specifies the last format in the list of available formats.

GetClipboardFormatName (wFormat, lpFormatName, nMaxCount) : nCopied

Purpose This function retrieves from the clipboard the name of the registered format specified by *wFormat*. The name is copied to the buffer pointed to by *lpFormatName*.

Parameters *wFormat* is a unsigned short integer value specifying the type of format to be retrieved. It must not specify any of the predefined clipboard formats.

lpFormatName is a long pointer to the buffer to receive the format name.

nMaxCount is an integer value specifying the maximum length in bytes of the string to be copied to the buffer.

Return Value *nCopied* is an integer value specifying the actual length of the string copied to the buffer. It is 0 if the requested format does not exist or is a predefined format.

SetClipboardViewer (*hWnd*) : *hWndNext*

Purpose This function adds the window specified by *hWnd* to the chain of windows that are notified (via the WM_DRAWCLIPBOARD message) whenever the contents of the clipboard have changed.

Parameters *hWnd* is a handle to the window to receive clipboard viewer chain messages.

Return Value *hWndNext* is the window handle of next window in the clipboard viewer chain. This handle should be saved in static memory and used in responding to clipboard viewer chain messages.

Notes Windows that are part of the clipboard viewer chain must respond properly to WM_DESTROY, WM_CHANGECBCHAIN, and WM_DRAWCLIPBOARD messages. See the description of these messages in Chapter 8 for details.

If an application wishes to remove itself from the clipboard viewer chain, it must call **ChangeClipboardChain**.

GetClipboardViewer () : *hWnd*

Purpose This function retrieves the window handle of the first window in the clipboard viewer chain.

Parameters None.

Return Value *hWnd* is a handle to the window currently responsible for displaying the clipboard. It is NULL if there is no viewer.

ChangeClipboardChain (*hWnd*, *hWndNext*) : *bRemoved*

Purpose This function removes the window specified by *hWnd* from the chain of clipboard viewers and makes the window specified by *hWndNext* the descendant of *hWnd*'s ancestor in the chain.

Parameters *hWnd* is a handle to the window to be removed from the chain. The handle must previously have been passed to **SetClipboardViewer**.

hWndNext is a handle to the window after *hWnd* in the clipboard viewer chain (the handle returned by **SetClipboardViewer**, unless the sequence was changed in response to a WM_CHANGECHAIN message.)

Return Value *bRemoved*, a Boolean value, is nonzero if *hWnd* is found and removed. Otherwise, *bRemoved* is zero.

2.11 Input Functions

This section describes the functions used to control input to a given window. An application receives input from the system mouse, keyboard, or timer. The input functions let an application disable input from these devices, take complete control of the devices, or define special actions to take when input from a device is received.

An application must acquire the input focus to receive input. Windows directs all keyboard input to the window with the input focus. Only one window can have the input focus at any given time.

The window with the input focus is either the active window (a tiled or popup style window) or a child of the active window. For example, a dialog box may be the active window, but a child window within the dialog box (such as an edit control) may have the focus.

The mouse capture functions let an application capture all input from the system mouse. Normally, an application only receives mouse input when the mouse cursor is in its window.

The timer functions let an application direct a system timer to pass it a message after a given amount of time has elapsed, and define a special action to take when the message arrives.

The input enable functions let an application enable or disable all mouse and keyboard input for a given window.

SetFocus (*hWnd*) : *hWndPrev*

Purpose This function assigns the input focus to the window specified by *hWnd*. The input focus directs all subsequent keyboard input to the given window. The window, if any, that previously had the input focus loses it. If *hWnd* is NULL, key-strokes will be ignored.

SetFocus sends a WM_KILLFOCUS message to the window losing the input focus and a WM_SETFOCUS message to the window getting the input focus. It also activates either the window that's receiving the focus or the parent of the window receiving the focus.

Parameters *hWnd* is a handle to the window to receive the keyboard input.

Return Value *hWndPrev* is a handle to the window that had the input focus. It is NULL if there is no such window.

Notes If a window is active but it doesn't have the focus (so no window has the focus), any key pressed will produce a WM_SYSKEYUP, WM_SYSKEYDOWN, or WM_SYSCHAR message. If the VK_MENU key is also pressed, the *lParam* parameter of the message will have bit 20000000 (hexadecimal) set. Otherwise, the messages that are produced do NOT have this bit set.

GetFocus () : *hWnd*

Purpose This function retrieves the handle of the window currently owning the input focus.

Parameters None.

Return Value *hWnd* is a window handle if the function is successful. Otherwise, it is NULL.

GetKeyState (*nVirtKey*) : *nState*

Purpose This function retrieves the state of the virtual key specified by *nVirtKey*. The state specifies whether the key is up, down, or toggled.

<i>Parameters</i>	<i>nVirtKey</i> is an integer value specifying a virtual key. If the desired virtual key is a letter or digit ("A" through "Z," "a" through "z," or "0" through "9"), <i>nVirtKey</i> must be set to the ASCII value of that character. For other keys, it must be one of the values given in Table 2.2.
<i>Return Value</i>	<i>nState</i> is an integer value specifying the state of the given virtual key. If the high-order bit is 1, the key is down. Otherwise, it is up. If the low-order bit is 1, the key is toggled. A toggle key, such as "Caps-Lock," is toggled if it has been pressed an odd number of times since the system was started. The key is untoggled if the low bit is 0.

Table 2.2
Virtual Keys

Key	Meaning
VK_LBUTTON	Left mouse button
VK_RBUTTON	Right mouse button
VK_MBUTTON	Middle mouse button
VK_BACK	Backspace key
VK_TAB	Tab key
VK_RETURN	Return key
VK_SHIFT	Shift key
VK_CONTROL	Control (CTRL) key
VK_MENU	Alternate (ALT) key
VK_CAPITAL	Caps Lock key
VK_INSERT	Insert key
VK_DELETE	Delete key
VK_CLEAR	Clear key
VK_ESCAPE	Escape (ESC) key
VK_HELP	Help key
VK_NUMPAD0	Numeric keypad 0
VK_NUMPAD1	Numeric keypad 1
VK_NUMPAD2	Numeric keypad 2
VK_NUMPAD3	Numeric keypad 3
VK_NUMPAD4	Numeric keypad 4
VK_NUMPAD5	Numeric keypad 5
VK_NUMPAD6	Numeric keypad 6
VK_NUMPAD7	Numeric keypad 7
VK_NUMPAD8	Numeric keypad 8
VK_NUMPAD9	Numeric keypad 9
VK_MULTIPLY	Multiply key
VK_ADD	Add key
VK_SEPARATOR	Separator key
VK_SUBTRACT	Subtract key
VK_DECIMAL	Decimal point key

Table 2.2 (*continued*)

Key	Meaning
VK_DIVIDE	Divide key
VK_SPACE	Spacebar
VK_PRIOR	Prior Screen key
VK_NEXT	Next Screen key
VK_END	End of document key
VK_HOME	Home key
VK_CANCEL	Cancel key
VK_PAUSE	Pause key
VK_LEFT	Left arrow key
VK_UP	Up arrow key
VK_RIGHT	Right arrow key
VK_DOWN	Down arrow key
VK_SELECT	Select key
VK_PRINT	Print key
VK_F1	Function 1 key
VK_F2	Function 2 key
VK_F3	Function 3 key
VK_F4	Function 4 key
VK_F5	Function 5 key
VK_F6	Function 6 key
VK_F7	Function 7 key
VK_F8	Function 8 key
VK_F9	Function 9 key
VK_F10	Function 10 key
VK_F11	Function 11 key
VK_F12	Function 12 key
VK_F13	Function 13 key
VK_F14	Function 14 key
VK_F15	Function 15 key
VK_F16	Function 16 key

SetCapture (*hWnd*) : *hWndPrev*

Purpose This function causes all subsequent mouse input to be sent to the window given by *hWnd*, regardless of the position of the mouse cursor.

Parameters *hWnd* is a handle to the window to receive the mouse input.

Return Value *hWndPrev* is the handle of the window that receives all mouse input prior to the **SetCapture** call. It is NULL if there is no such window.

ReleaseCapture()

Purpose This function releases mouse input and restores normal input processing.

Parameters None.

Return Value None.

SetTimer(hWnd, nIDEvent, wElapse, lpTimerFunc) : nIDNewEvent

Purpose This function creates a system timer event identified by *nIDEvent*. When a timer event occurs, Windows passes a WM_TIMER message to the user-supplied function specified by *lpTimerFunc*. The function can then process the event. A NULL value for *lpTimerFunc* causes WM_TIMER messages to be placed in the application queue.

Parameters *hWnd* is a handle to a window.

nIDEvent is an integer value identifying the timer event. If *hWnd* is NULL, the *nIDEvent* parameter is ignored. Instead, **SetTimer** returns a unique ID that can be used later with **KillTimer**.

wElapse is an unsigned short integer value specifying the elapsed time in milliseconds between timer events.

lpTimerFunc is a long pointer to the function to be notified when the timer event takes place. If *lpTimerFunc* is NULL, the WM_TIMER message is placed in the application queue; the *hwnd* member of the MSG structure contains the *hWnd* parameter given in the **SetTimer** call.

Return Value *nIDNewEvent* is an integer value specifying the new timer event. If *hWnd* is NULL, **SetTimer** returns a unique ID. If *hWnd* is not NULL, the return value is the same as the *nIDEvent* parameter. The return value is 0 if the timer event is not set.

Notes The *lpTimerFunc* function must have the form:

functionname(hWnd, wMsg, nIDEvent, dwTime)

where *hWnd* is a handle to the window associated with the timer event, *wMsg* is the WM_TIMER message, *nIDEvent* is a short integer value specifying the timer's ID, and *dwTime* is a long unsigned integer specifying the current system time. The function can carry out any desired tasks.

KillTimer (*hWnd*, *nIDEvent*) : *bKilled*

Purpose This function kills the timer event identified by *hWnd* and *nIDEvent*. Any pending WM_TIMER messages associated with the timer are removed from the message queue.

Parameters *hWnd* is the handle of the window associated with the given timer event.

nIDEvent is an integer value identifying the timer event to be killed. This must be a valid event ID returned by **SetTimer** or 0. If it is 0, **KillTimer** removes an arbitrary timer event.

Return Value *bKilled*, a Boolean value, is nonzero if the event is killed. It is zero if it cannot find the specified timer event.

EnableWindow (*hWnd*, *bEnable*) : *bDone*

Purpose This function enables and disables mouse and keyboard input to the specified window. When input is disabled, input such as mouse clicks and key presses are ignored by the window. When enabled, all input is processed.

EnableWindow enables mouse and keyboard input to a window if *bEnable* is nonzero and disables it if *bEnable* is zero.

Parameters *hWnd* is a handle to a window.

bEnable is a Boolean value specifying whether or not the given window is to be enabled for input.

Return Value *bDone*, a Boolean value, is nonzero if the window is enabled or disabled as specified. It is zero if an error occurs.

Notes A window must be enabled before it can be activated. For example, if an application is displaying a modeless dialog box and has disabled its main window, the main window must be enabled before the dialog box is destroyed. Otherwise, another window (or no window, if there are not other windows) will get the input focus and be activated.

If a child window is disabled, it is ignored when Windows tries to determine which window should get mouse messages.

Initially, all windows are enabled by default. **EnableWindow** must be used to disable a window explicitly.

IsWindowEnabled (*hWnd*) : *bEnabled*

Purpose This function specifies whether or not the specified window is enabled for mouse and keyboard input.

Parameters *hWnd* is a handle to a window.

Return Value *bEnabled*, a Boolean value, is nonzero if the window is enabled. Otherwise, it is zero.

Notes A child window only receives input if it is both enabled and visible.

2.12 Menu Functions

This section describes the functions used to create, modify, and destroy menus. Applications access menus through menu handles, which are assigned by Windows.

Two types of menus can be distinguished: top-level (or “menu bar”) menus, and popup menus (also called “drop-down” or “pull-down” menus.) A top-level menu consists of menu items displayed in a window’s menu bar. A popup menu is a menu associated with a top-level menu item; when the user selects that menu item, the popup menu appears. The popup menu has its own unique menu handle (as does each top-level menu).

Each item on a top-level menu has either a command value or a popup menu associated with it. If the item has no popup menu, the command value is sent to the application in a WM_COMMAND message. If the item has a popup menu, that menu is displayed. If the user selects an item from the system menu, the command value for that item is sent in a WM_SYSCOMMAND message.

Individual menu items can be grayed, disabled, enabled, checked, or unchecked (except that top-level menu items cannot be checked.)

SetMenu (*hWnd*, *hMenu*) : *bSet*

Purpose This function sets the given window’s menu to *hMenu*. If *hMenu* is NULL, the window’s current menu is removed. **SetMenu** causes the window to be redrawn to reflect the menu change.

Parameters *hWnd* is a handle to the window whose menu is to be changed.

hMenu is a handle to the new menu.

Return Value *bSet*, a Boolean value, is nonzero if the menu is changed. Otherwise, it is zero.

GetMenu (*hWnd*) : *hMenu*

Purpose This function retrieves a handle to the menu of the specified window.

Parameters *hWnd* is a handle to the window whose menu is to be examined.

Return Value *hMenu* is a handle to a menu. It is NULL if the given window has no menu.

CreateMenu () : *hMenu*

Purpose This function creates a menu. The menu is initially empty, but can be filled with menu items by using the **ChangeMenu** function.

Parameters None.

Return Value *hMenu* is a handle to the newly-created menu. It is NULL if the menu cannot be created.

DestroyMenu (*hMenu*) : *bDestroyed*

Purpose This function destroys the menu specified by *hMenu* and frees any memory that it occupied.

Parameters *hMenu* is a handle to the menu to be destroyed.

Return Value *bDestroyed*, a Boolean value, is nonzero if the menu is destroyed. Otherwise, it is zero.

ChangeMenu (*hMenu*, *wIDChangeItem*, *lpNewItem*, *wIDNewItem*, *wChange*) : *bChanged*

Purpose This function appends, inserts, deletes, or modifies a menu item in the menu given by *hMenu*. The *wIDChangeItem*, *lpNewItem*, *wIDNewItem*, and *wChange* parameters define which item to change and how to change it.

Parameters *hMenu* is a handle to the menu to be changed.

wIDChangeItem is an unsigned short integer value identifying the item to be changed. If the MF_BYPOSITION flag is specified, *wIDChangeItem* gives the position of the menu item to be changed (the first item is at position 0). If MF_BYCOMMAND is specified instead, *wIDChangeItem* is the menu item ID. The menu item ID can specify an sub-

menu item (that is, an item in a popup menu associated with an item of *hMenu*). If neither flag is given, the default is MF_BYCOMMAND. When MF_INSERT is used, *wIDChangeItem* identifies the item before which the new item is to be inserted. When MF_APPEND is used, *wIDChangeItem* is NULL.

lpNewItem is either a handle to a bitmap (when MF_BITMAP is specified in the *wChange* parameter), a handle to a menu (when MF_POPUP is specified) or a long pointer to a character string (when MF_STRING is specified). The default is MF_STRING. A NULL value for *lpNewItem* creates a horizontal break (the same effect as using the MF_SEPARATOR flag).

wIDNewItem is an unsigned short integer value identifying the new menu item. If the MF_BYPOSITION flag is specified, *wIDNewItem* gives the position of the new menu item (the first item is at position 0). If MF_BYCOMMAND is specified instead, *wIDNewItem* is the new menu item ID. If neither flag is given, the default is MF_BYCOMMAND. When the MF_POPUP flag is used, *wIDNewItem* is a handle to the popup menu.

wChange is an unsigned short integer value specifying how to change the menu. It consists of one or more of the values given in Table 2.3. These values are combined using the bitwise OR operator.

Return Value *bChanged*, a Boolean value, is nonzero if the change is successful. Otherwise, it is zero.

Notes If a top-level menu item is changed (i.e., a menu item appearing in the menu bar at the top of an application's window), the **DrawMenuBar** function should be called to redraw the menu bar.

ChangeMenu is useful for inserting, deleting, changing, and appending menu items, and setting the attributes of the new or changed items. However, to change the attributes of existing menu items, it is much faster to use **CheckMenuItem** and **EnableMenuItem**.

Table 2.3
ChangeMenu Flags

Flag	Meaning
MF_CHANGE	Change or replace the specified item.
MF_INSERT	Insert a new item just before the specified item.
MF_APPEND	Append the new item to the end of the menu.
MF_DELETE	Delete the item.
MF_BYPOSITION	<i>wIDChangeItem</i> gives the position of the menu item to be changed rather than an ID number.
MF_BYCOMMAND	<i>wIDChangeItem</i> gives the menu item ID number (default).
MF_GRAYED	Disable and gray the item to show that it cannot be selected.
MF_ENABLED	Enable the item, allowing it to be selected (default).
MF_DISABLED	Disable the item without changing its appearance (it cannot be selected).
MF_CHECKED	Place a checkmark next to the item (popup menu item only).
MF_UNCHECKED	Do not place a checkmark next to the item (default).
MF_MENUBREAK	For static (menu bar) menus, place the item on a new line. For popup menus, place the item in a new column, with no dividing line between the columns.
MF_MENU BARBREAK	Same as MF_MENUBREAK, except that for popup menus, separate the new column from the old column with a vertical divider line.
MF_SEPARATOR	Draw a horizontal dividing line. Can only be used in popup menus. Cannot be enabled, checked, grayed, or highlighted. <i>lpNewItem</i> and <i>wIDNewItem</i> are ignored.
MF_BITMAP	Use a bitmap as the item. The low-order word of <i>lpItem</i> is a handle specifying the bitmap.

Table 2.3 (continued)

Flag	Meaning
MF_STRING	Use a string as the item (default). <i>lpNewItem</i> is a long pointer to a null-terminated ASCII string.
MF_POPUP	Associates a popup menu with a menu item. The <i>wIDNewItem</i> parameter is a handle to the menu. Can only be used with top-level menu items.
<i>Note</i>	When the user selects a menu item it is automatically highlighted. To control highlighting of menu items, use the HiliteMenuItem function.

DrawMenuBar (*hWnd*)

Purpose This function redraws the menu bar. If a menu bar is changed *after* Windows has created the window and drawn the menu bar for the first time, this function should be called to draw the changed menu bar.

Parameters *hWnd* is a handle to the window whose menu needs redrawing.

Return Value None.

CheckMenuItem (*hMenu*, *wIDCheckItem*, *wCheck*) : *bOldCheck*

Purpose This function places or removes checkmarks from menu items in the popup menu specified by *hMenu*. The *wIDCheckItem* parameter specifies the item to be modified.

Parameters *hMenu* is a handle to a menu.

wIDCheckItem is an unsigned short integer value identifying the menu item to be checked.

wCheck, an unsigned short integer value, consists of one or more of the following values, combined with the bitwise OR operator:

Value	Meaning
MF_CHECKED	Checkmark is added.
MF_UNCHECKED	Checkmark is removed.
MF_BYPOSITION	The <i>wIDCheckItem</i> parameter gives the position of the menu item (the first item is at position zero.)
MF_BYCOMMAND	The <i>wIDCheckItem</i> parameter gives the menu item ID (MF_BYCOMMAND is the default).

Return Value *bOldCheck*, a Boolean value, gives the previous state of the item. It is either MF_CHECKED or MF_UNCHECKED.

Notes The *wIDCheckItem* parameter may identify a submenu item as well as a menu item. No special steps are required to check a submenu item.

Top-level menu items cannot be checked.

EnableMenuItem (*hMenu*, *wIDEnableItem*, *wEnable*) : *bEnabled*

Purpose This function enables, disables, or grays a menu item.

Parameters *hMenu* is a handle to a menu.

wIDEnableItem is an unsigned short integer value identifying the menu item to be checked. The *wIDEnableItem* parameter can specify submenu items as well as menu items. No special steps are required to enable a submenu item.

wEnable is an unsigned short integer value specifying the action to take. It can be one or more of the following values, combined with the bitwise OR operator:

Value	Meaning
MF_GRAYED	Menu item is grayed.
MF_ENABLED	Menu item is enabled.
MF_DISABLED	Menu item is disabled.

MF_BYPOSITION	The <i>wIDEnableItem</i> parameter gives the position of the menu item (the first item is at position zero.)
MF_BYCOMMAND	The <i>wIDEnableItem</i> parameter gives the menu item ID (MF_BYCOMMAND is the default).

Return Value *bEnabled* is a Boolean value specifying the previous state of the menu item.

Notes To disable or enable input to a menu bar, see the WM_SYSCOMMAND message in Chapter 8.

HiliteMenuItem (*hWnd*, *hMenu*, *wIDHiliteItem*, *wHilite*) : *bHilited*

Purpose This function highlights or removes the highlighting from a top-level (menu bar) menu item.

Parameters *hWnd* is a window handle.

hMenu is a handle to a top-level menu.

wIDHiliteItem is either the control ID of the menu item or the offset of the menu item in the menu, depending on the value of the *wHilite* parameter.

wHilite is an unsigned short integer value containing one or more of the following values. The values can be combined using the bitwise OR operator.

Value	Meaning
MF_BYCOMMAND	Interpret the <i>wIDHiliteItem</i> parameter as the menu item ID (the default interpretation).
MF_BYPOSITION	Interpret the <i>wIDHiliteItem</i> parameter as an offset.
MF_HILITE	Highlight the item. If this flag is not given highlighting is removed from the item.
MF_UNHILITE	Remove highlighting from the item.

Return Value *bHilited*, a Boolean value, is nonzero if the item is highlighted. Otherwise, it is zero.

Notes The MF_HILITE and MF_UNHILITE flags can be used only with the **HiliteMenuItem** function; they cannot be used with **ChangeMenu**.

GetSubMenu (*hMenu*, *nPos*) : *hPopupMenu*

Purpose This function retrieves the menu handle of a popup menu.

Parameters *hMenu* is a handle to a menu.

nPos is a short integer value specifying the position in the given menu of the popup menu. Position values start at 0 for the first menu item. The popup menu's ID cannot be used in this function.

Return Value *hPopupMenu* is a handle to the given popup menu. It is NULL if no popup menu exists at the given position.

GetSystemMenu (*hWnd*, *bRevert*) : *hSysMenu*

Purpose This function allows access to the system menu for copying and modification.

Parameters *hWnd* is a handle to the window that will own a copy of the system menu.

bRevert is a Boolean value that specifies the action to be taken. If *bRevert* is zero, **GetSystemMenu** returns a handle to a copy of the system menu. This copy is initially identical to the system menu, but can be modified.

If *bRevert* is nonzero, **GetSystemMenu** destroys the possibly modified copy of the system menu (if there is one) belonging to the specified window and returns a handle to the original, unmodified version of the system menu.

Return Value *hSysMenu* is a handle to the system menu if *bRevert* is nonzero. If *bRevert* is zero, *hSysMenu* is a handle to a copy of the system menu.

Notes Any window that does not use **GetSystemMenu** to make its own copy of the system menu gets the standard system menu.

The handle returned by **GetSystemMenu** can be used with the **ChangeMenu** function to change the system menu.

The system menu initially contains five items, with the following ID values:

SC_SIZE
 SC_MOVE
 SC_ICON
 SC_ZOOM
 SC_CLOSE

The following system menu ID values are predefined and available for use on the application's system menu:

Value	Meaning
SC_NEXTWINDOW	Make the next window active.
SC_PREVWINDOW	Make the previous window active.
SC_VSCROLL	Assume a movement on the vertical scroll bar.
SC_HSCROLL	Assume a movement on the horizontal scroll bar.
SC_MOUSEMENU	Assume a mouse-activated menu.
SC_KEYMENU	Assume a keyboard-activated menu.

Menu items on the system menu send WM_SYSCOMMAND messages. All predefined system menu items have ID numbers greater than F000 (hexadecimal). If an application adds commands to the system menu, it should use ID numbers less than F000.

Windows automatically grays items on the standard system menu, depending on the situation. The application can carry out its own checking or graying by responding to the WM_INITMENU message, which is sent before any menu is pulled down.

**GetMenuItemString (*hMenu*, *wIDItem*, *lpString*, *nMaxCount*, *wFlag*)
 : *nCopied***

Purpose This function copies the label of the specified menu item into *lpString*.

Parameters *hMenu* is a handle to the menu.

wIDItem is either the ID of the menu item (from the resource file) or the offset of the menu item in the menu, depending on the value of *wFlag*.

lpString is a long pointer to the buffer to receive the label.

nMaxCount is a short integer value specifying the maximum length of the label to be copied. If the label is longer than *nMaxCount*, the extra characters are truncated.

wFlag, an unsigned short integer value, is either MF_BYPOSITION or MF_BYCOMMAND.

MF_BYPOSITION causes the *wIDItem* parameter to be interpreted as an offset, while MF_BYCOMMAND causes *wIDItem* to be interpreted as the control ID of the menu item.

Return Value *nCopied* is a short integer value specifying the number of bytes copied to the buffer.

2.13 Window Painting Functions

This section describes the functions used to prepare the client area of a window for graphics operations.

GetDC (*hWnd*) : *hDC*

Purpose This function retrieves the display context for the client area of the specified window. The display context can be used in subsequent GDI functions to draw in the client area.

If the window class of the window specified by *hWnd* does not have CS_OWNDC style, **GetDC** must be called before any drawing is done in the window, and **ReleaseDC** must be called when drawing is finished. **GetDC** returns a handle to the default display context, which has the following characteristics:

Attribute	Default Selection
Brush	WHITE_BRUSH
Pen	BLACK_PEN
Bitmap	No default
Font	SYSTEM_FONT
Current Pen Position	(0,0)
Brush Origin	(0,0)
Relabs Flag	ABSOLUTE
Text Color	Black
Background Color	White
Background Mode	OPAQUE
Drawing Mode	R2_COPYPEN
Stretching Mode	BLACKONWHITE
Mapping Mode	MM_TEXT
Intercharacter Spacing	0
Polygon Filling Mode	ALTERNATE
Color Table	Entries are set to represent an even distribution of the full range of available colors. The exact default values depend on the device. For example, raster devices have black as the first entry, white as the last; printers have white as the first entry, black as the last.
Clipping Region	The whole display surface
Window Origin	(0, 0)
Window Extents	(1, 1)
Viewport Origin	(0, 0)
Viewport Extents	(1, 1)

The attributes of the display context can be changed using the selection functions and display context attribute functions described in Chapter 3.

Normally, all state information (currently selected brush, map mode, clipping region, etc.) that was selected into the display context after a **GetDC** call is lost after a **ReleaseDC** call. If a window belongs to a class that has CS_OWNDC style, however, the state information of a display context is preserved, meaning that attributes such as clipping region, currently selected brush, etc., remain selected until the application specifically makes a change. The handle to the display context returned by **GetDC** remains valid, even if **ReleaseDC** is called.

If the window specified by *hWnd* belongs to a class with CS_CLASSDC style, all windows in the same class use the same display context. Each call to **GetDC** with a different window from the same class returns the same display context handle. State information for the display context is retained, except that the clipping region, window origin, and window extents are reset to the default values each time **GetDC** is called for a new window. The sequence of getting and releasing the display context must be followed for each window in the class; it is an error to call **GetDC** for two different windows in the same class without first calling **ReleaseDC** to release the display context for the first window.

Parameters *hWnd* is a handle to a window. If *hWnd* is NULL, a saved display context is returned.

Return Value If the function is successful, *hDC* is a handle to the display context for the given window's client area. Otherwise, it is NULL.

Notes Only five display contexts are available in the Windows systems at a given time (not counting display contexts allocated by giving a window class CS_CLASSDC or CS_OWNDC style.) **ReleaseDC** should be called promptly after finishing with a display context to free the display context for use by other windows and applications.

GetWindowDC (*hWnd*) : *hDC*

Purpose This function retrieves the display context for the entire window, including caption bar, menus, and scroll bars. The origin of the display context is the upper left corner of the window, which corresponds to the upper left corner of the caption bar.

<i>Parameters</i>	<i>hWnd</i> is a handle to a window.
<i>Return Value</i>	<i>hDC</i> is a handle to the display context for the given window if the function is successful. Otherwise, it is NULL.
<i>Notes</i>	Using the display context returned by GetWindowDC , it is possible to draw anywhere in the window, even in the caption bar, menus, and scroll bars. Use GetSystemMetrics to get the dimensions of the caption bar, menu areas, and scroll bars, or GetDC to get the display context of the window's client area.
	The display context returned by <i>GetWindowDC</i> should be released using ReleaseDC when drawing is finished. Unlike GetDC , state information (such as the currently selected brush or map mode) is not preserved when ReleaseDC is called, even if the class to which the specified window belongs has CS_OWNDC or CS_CLASSDC style.

ReleaseDC (*hWnd, hDC*) : *nReleased*

<i>Purpose</i>	This function is used to release a display context when an application is finished drawing in it.
<i>Parameters</i>	<i>hWnd</i> is a handle to a window. <i>hDC</i> is a handle to the display context.
<i>Return Value</i>	<i>nReleased</i> , a short integer value, is 1 if the display context is released. Otherwise, it is 0.
<i>Notes</i>	For each call to GetDC or GetWindowDC , there should be a matching call to ReleaseDC . Only five display contexts are available in the Windows systems at a given time (not counting display contexts allocated by giving a window class CS_CLASSDC or CS_OWNDC style). ReleaseDC should be called promptly after finishing with a display context to free the display context for use by other windows and applications.

BeginPaint (*hWnd, lpPaint*) : *hDC*

<i>Purpose</i>	This function prepares the given window for painting and fills the paint structure pointed to by <i>lpPaint</i> with information about the painting, such as the area of the screen that needs redrawing. If necessary, a WM_ERASEBKGND message is sent to the window.
----------------	--

Parameters *hWnd* is a handle to the window to be repainted.
 lpPaint is a long pointer to a data structure having
 PAINTSTRUCT type.

Return Value *hDC* is the display context for the specified window. The
 clipping region of the display context includes only the part
 of the client area that needs redrawing.

Notes If the caret is in the area to be painted, **BeginPaint**
 automatically hides the caret to prevent overwriting.

EndPaint (*hWnd*, *lpPaint*)

Purpose This function marks the end of repainting by the window
 and is required after each **BeginPaint** call.

Parameters *hWnd* is a handle to the window that has been repainted.
 lpPaint is a long pointer to a data structure having
 PAINTSTRUCT type.

Return Value None.

Notes If the caret was hidden by the **BeginPaint** function,
 EndPaint restores the caret to the screen.

UpdateWindow (*hWnd*)

Purpose This function is used to ensure that a window's appearance
 is up-to-date. It notifies the application when parts of a
 window need redrawing after the window is changed (by
 resizing, scrolling, etc.) **UpdateWindow** also checks child
 windows of the specified window to determine if they need
 updating.

UpdateWindow sends WM_PAINT messages directly to
the window function of the specified window, bypassing the
application queue, with the result that the window has been
repainted by the time **UpdateWindow** returns.

If no repainting is necessary, no message is sent.

Parameters *hWnd* is a handle to the window to be updated.

Return Value None.

GetUpdateRect (*hWnd*, *lpRect*, *bErase*) : *bUpdate*

Purpose This function returns the rectangle that bounds the region of the window that needs updating.

Parameters *hWnd* is a handle to the window to be checked.

lpRect is a long pointer to a **RECT** data structure.

GetUpdateRect fills the structure with the coordinates of the rectangle bounding the area that needs updating. The coordinates are 0,0,0,0 (null) if no updating is necessary.

bErase is a Boolean value specifying whether **GetUpdateRect** should send a WM_ERASEBKGND message to erase the background of the update area. If the update region is not null and *bErase* is nonzero, the message is sent.

Return Value *bUpdate*, a Boolean value, is nonzero if any area of the window needs updating. If the window is up-to-date, *bUpdate* is zero.

Note The rectangle coordinates provided by **GetUpdateRect** are the same as the coordinates given in the **PAINTSTRUCT** data structure after **BeginPaint** is called.

InvalidateRect (*hWnd*, *lpRect*, *bErase*)

Purpose This function marks for repainting a rectangular area within the specified window. The *lpRect* parameter defines the rectangle, in client coordinates, enclosing the area to be repainted. If *bErase* is nonzero, the contents of the rectangle are erased when marked; if zero, the contents remain unchanged.

Parameters *hWnd* is the handle of the window to be repainted.

lpRect is a long pointer to a data structure having **RECT** type that specifies the rectangle (in client coordinates) to be repainted. If *lpRect* is NULL, the entire window is invalidated.

bErase is a Boolean value specifying whether or not the given rectangle needs to be erased.

Return Value None.

Notes **InvalidateRect** does not cause the given rectangle to be repainted immediately. It sends a WM_PAINT message, and the application eventually repaints the area in response to the message. If **InvalidateRect** is called more than once for the same window before **GetMessage**, **PeekMessage**, or **UpdateWindow** is called, the rectangles given in each

call are combined into a single update region. The WM_PAINT message that an application eventually receives (by calling **GetMessage**, **PeekMessage**, or **UpdateWindow**) gives the entire update region.

InvalidateRgn (*hWnd*, *hRgn*, *bErase*)

Purpose This function marks for repainting a region within the display screen of the specified window. The *hRgn* parameter defines the region to be repainted. If *bErase* is nonzero, the contents of the region are erased when marked; if zero, the contents remain unchanged.

Parameters *hWnd* is the handle of the window to be repainted.
hRgn is a handle to a region, assumed to be in client coordinates.

bErase is a Boolean value specifying whether or not the screen is to be erased.

Return Value None.

Notes **InvalidateRgn** has the same effect as **InvalidateRect**, except that a handle to a region is passed instead of a pointer to a rectangle.

InvalidateRgn does not cause the given region to be repainted immediately. It sends a WM_PAINT message, and the application eventually repaints the area in response to the message. If **InvalidateRgn** is called more than once for the same window before **GetMessage**, **PeekMessage**, or **UpdateWindow** is called, the regions given in each call are combined into a single update region. The WM_PAINT message that an application eventually receives (by calling **GetMessage**, **PeekMessage**, or **UpdateWindow**) gives the entire update region.

ValidateRect (*hWnd*, *lpRect*)

Purpose This function releases from repainting a previously marked rectangular area of the display screen in the window specified by *hWnd*. The *lpRect* parameter specifies the dimensions, in client coordinates, of the rectangular area to be released. If *lpRect* is NULL, the entire window is validated.

Parameters *hWnd* is a handle to the window containing the area.
lpRect is a long pointer to a data structure having **RECT** type. The rectangle coordinates are assumed to be client coordinates.

Return Value None.

Notes **ValidateRect** or **ValidateRgn** should be called every time part of a window is updated. This prevents the screen from being redrawn twice.

ValidateRgn (*hWnd*, *hRgn*)

Purpose This function releases from repainting a previously marked region in the display screen of the window specified by *hWnd*. The *hRgn* parameter specifies the region to be released. If *hRgn* is NULL, the entire window is released.

Parameters *hWnd* is a handle to the window containing the area.

hRgn is a handle to a region, assumed to be in client coordinates.

Return Value None.

Notes **ValidateRgn** is the same as **ValidateRect**, except that a handle to a region is passed instead of a pointer to a rectangle.

ValidateRect or **ValidateRgn** should be called every time part of a window is updated. This prevents the screen from being redrawn twice.

2.14 Scrolling Functions

This section describes the functions used to scroll the contents of a window's client area and control the window's scroll bars.

ScrollWindow (*hWnd*, *XAmount*, *YAmount*, *lpRect*, *lpClipRect*)

Purpose This function scrolls a window by moving the contents of the window's client area *XAmount* units along the screen's x-axis and *YAmount* units along the y-axis. The contents of the window scroll left if *XAmount* is positive and right if it is negative. The contents of the window scroll down if *YAmount* is positive and up if it is negative.

<i>Parameters</i>	<p><i>hWnd</i> is a handle to a window.</p> <p><i>XAmount</i> is the amount (in device units) to scroll in the <i>X</i> direction.</p> <p><i>YAmount</i> is the amount (in device units) to scroll in the <i>Y</i> direction.</p> <p><i>lpRect</i> is a long pointer to a data structure having RECT type. The rectangle specifies what portion of the window's client area should be scrolled. If <i>lpRect</i> is NULL, the entire client area is scrolled, and the caret position is adjusted accordingly.</p> <p><i>lpClipRect</i> is a long pointer to a data structure having RECT type. Only bits inside this rectangle are affected by the scroll. If <i>lpClipRect</i> is NULL, the entire window is affected.</p>
<i>Return Value</i>	None.
<i>Notes</i>	<p>If the caret is in the window being scrolled, ScrollWindow automatically hides the caret to prevent overwriting, then restores the caret after the scroll is finished.</p> <p>The area "uncovered" by ScrollWindow is not repainted by ScrollWindow, but is combined into the window's update region. The application will eventually receive a WM_PAINT message notifying it that the region needs repainting. To repaint the uncovered area at the same time the scrolling is done, call UpdateWindow immediately after calling ScrollWindow.</p> <p>If <i>lpRect</i> is NULL, then the positions of any child windows in the window are offset by <i>XAmount</i> and <i>YAmount</i>, and any invalid (unpainted) areas in the window are offset as well. ScrollWindow is somewhat faster when <i>lpRect</i> is NULL.</p> <p>If <i>lpRect</i> is not NULL, then the positions of child windows are NOT changed, nor are invalid areas in the window offset. To prevent updating problems when <i>lpRect</i> is not NULL, calling UpdateWindow to repaint the window before calling ScrollWindow is recommended.</p>

SetScrollPos (*hWnd*, *nBar*, *nPos*, *bRedraw*) : *nOldPos*

Purpose This function sets the current position of a scroll bar elevator to *nPos*, and if specified, redraws the scroll bar to reflect the new position.

Parameters *hWnd* is a handle to the window whose scroll bar is to be set. *nBar* is one of the following short integer values, specifying which scroll bar to set:

Value	Meaning
SB_VERT	Set the vertical scroll bar position.
SB_HORZ	Set the horizontal scroll bar position.
SB_CTL	Set a scroll bar control; in this case, the <i>hWnd</i> parameter specifies the child window handle of the scroll bar control.

nPos is a short integer value specifying the new position. It must be within the range specified by the **SetScrollRange** function.

bRedraw is a Boolean value specifying whether or not the scroll bar should be redrawn to reflect the change. If *bRedraw* is nonzero, the scroll bar is redrawn. If zero, it is not redrawn. Setting the *bRedraw* parameter to zero is useful whenever the scroll bar will be redrawn by a subsequent call to another function.

Return Value *nOldPos* is a short integer value specifying the previous position of the scroll bar elevator.

GetScrollPos (*hWnd*, *nBar*) : *nPos*

Purpose This function retrieves the current position of a scroll bar elevator.

Parameters *hWnd* is a handle to the window whose scroll bar is to be examined.

nBar is a short integer value specifying which scroll bar to examine. If *nBar* is SB_VERT, the function examines the vertical scroll bar; if *nBar* is SB_HORZ, it examines the horizontal scroll bar. The SB_CTL value indicates that the

operation is to take place on a scroll bar control rather than on a standard scroll bar; in this case, *hWnd* specifies the child window handle of a scroll bar control.

Return Value *nPos* is a short integer value specifying the current position of the scroll bar elevator.

SetScrollRange (*hWnd*, *nBar*, *nMinPos*, *nMaxPos*, *bRedraw*)

Purpose This function sets minimum and maximum position values for the scroll bar of the specified window. It can also be used to add or remove a window's scroll bar.

The default scroll range is 0 to 100, but this range can be reset to any useful values. For example, a spreadsheet program with 255 rows can set the vertical scroll range to 1 to 255.

Parameters *hWnd* is a handle to the window whose bar is to be set.

nBar is one of the following short integer values, specifying which scroll bar to set:

Value	Meaning
SB_VERT	Set the vertical scroll bar position.
SB_HORZ	Set the horizontal scroll bar position.
SB_CTL	Set a scroll bar control; in this case, the <i>hWnd</i> parameter specifies the child window handle of a scroll bar control.

nMinPos and *nMaxPos* are short integer values specifying the minimum and maximum scrolling positions, respectively. If *nMinPos* is equal to *nMaxPos*, the scroll bar is removed.

bRedraw is a Boolean value specifying whether or not the scroll bar should be redrawn to reflect the change. If *bRedraw* is nonzero, the scroll bar is redrawn. If zero, it is not redrawn.

Return Value None.

Notes When a scroll bar is being hidden or removed, the window is always updated to reflect the change, regardless of the value of the *bRedraw* parameter.

If a scroll bar is removed, you can restore it by calling **SetScrollRange** with unequal minimum and maximum values.

If **SetScrollRange** immediately follows **SetScrollPos**, *bRedraw* in **SetScrollPos** should be set to zero to prevent the scroll bar from being drawn twice.

GetScrollRange (*hWnd*, *nBar*, *lpMinPos*, *lpMaxPos*)

Purpose This function retrieves the current minimum and maximum scroll bar positions for the given scroll bar and saves the values in the locations specified by *lpMinPos* and *lpMaxPos*.

Parameters *hWnd* is a handle to the window to be examined.

nBar is one of the following short integer values, specifying which scroll bar to set:

Value	Meaning
SB_VERT	Set the vertical scroll bar position.
SB_HORZ	Set the horizontal scroll bar position.
SB_CTL	Set a scroll bar control; in this case, the <i>hWnd</i> parameter specifies the control.

lpMinPos is a long pointer to the short integer variable to receive the minimum position.

lpMaxPos is a long pointer to the short integer variable to receive the maximum position.

Return Value None.

2.15 Property List Functions

This section describes the functions used to access the property list of a window.

SetProp (*hWnd*, *lpString*, *hData*) : *bSet*

Purpose This function copies a character string and a data handle to the property list of the window specified by *hWnd*. The string, pointed to by *lpString*, serves as a label for the data handle *hData*. The data handle can be retrieved from the property list at any time by using the **GetProp** function and specifying the string.

Parameters *hWnd* is a handle to a window.

lpString is either a long pointer to a character string or an atom identifying a string. If the low-order word of *lpString* is in the range 0 to BFFF (hexadecimal) and the high-order word is zero, it is treated as an atom.

hData is a handle to an arbitrary data structure.

Return Value *bSet*, a Boolean value, is nonzero if the function is successful. Otherwise, it is zero.

Note The application must free any data handles in a window's property list before it destroys the window.

GetProp (*hWnd*, *lpString*) : *hData*

Purpose This function searches the window's property list for a data handle associated with the string pointed to by *lpString*.

Parameters *hWnd* is a handle to a window.

lpString is either a long pointer to a character string or an atom identifying a string. If the low-order word of *lpString* is in the range 0 to BFFF (hexadecimal) and the high-order word is zero, it is treated as an atom.

Return Value If the function is successful, *hData* is a handle to an arbitrary data structure. Otherwise, it is NULL.

RemoveProp (*hWnd*, *lpString*) : *hData*

Purpose This function removes the specified string from the window property list and retrieves the data handle named by that string so that the data associated with the handle can be freed.

Parameters *hWnd* is a handle to a window.

lpString is a long pointer to a null-terminated ASCII string or an atom identifying a string. If the value is in the range 0 to BFFF (hexadecimal), it is treated as an atom.

Return Value *hData* is a handle to an a data structure named by the string if the function is successful. Otherwise, it is NULL.

EnumProps (*hWnd*, *lpEnumFunc*) : *nResult*

Purpose This function enumerates all the properties of the specified window. The properties are passed, one by one, to the application-supplied function specified by *lpEnumFunc*. The function continues until the last property is enumerated or the function returns zero.

Parameters *hWnd* is a handle to a window.

lpEnumFunc is a long pointer to a application-supplied function.

Return Value *nResult*, a short integer value, is equal to the last value returned by the application-supplied function.

Notes The application-supplied function must have the form

functionname (hWnd, lpString, hData) : bContinue

where *hWnd* is a handle to a window and *hData* is an arbitrary handle.

The function can carry out any desired task. It must return a nonzero value to continue enumeration. It can return zero to stop the enumeration.

The *lpString* parameter is a far pointer to a buffer on the stack. If the stack segment is the same as the data segment (a STACKSIZE statement is included in the application's .DEF file), and the data segment moves as a result of local memory allocation, this far pointer will be invalidated. Therefore, if SS==DS, the application-supplied function is declared as follows:

functionname (hWnd, nDummy, pString, hData) : bContinue

where *nDummy* is an unused short integer variable and *pString* is a short pointer to a string. The other parameters and the return value are the same as described above.

2.16 Window Attribute Functions

This section describes the functions used to access and modify the attributes of a window.

SetWindowText (*hWnd*, *lpString*)

Purpose This function sets the given window's caption title (if it has one) to the string pointed to by *lpString*. If *hWnd* is a handle to a control, **SetWindowText** sets the text within the control instead of the caption.

Parameters *hWnd* is a handle to the window or control whose text is to be changed.

lpString is a long pointer to a null-terminated string.

Return Value None.

GetWindowText (*hWnd*, *lpString*, *nMaxCount*) : *nCopied*

Purpose This function copies the given window's caption title (if it has one) into the buffer pointed to by *lpString*. If *hWnd* is a handle to a control, **GetWindowText** copies the text within the control instead of the caption.

Parameters *hWnd* is a handle to the window or control whose caption or text is to be copied.

lpString is a long pointer to the buffer to receive the copied string.

nMaxCount is a short integer value specifying the maximum number of characters to be copied to the buffer.

Return Value *nCopied* is a short integer value specifying the length of the copied string. It is 0 if the window has no caption.

GetWindowTextLength (*hWnd*) : *nLength*

Purpose This function returns the length of the given window's caption title. If *hWnd* is a handle to a control, **GetWindowTextLength** returns the length of the text within the control instead of the caption.

Parameters *hWnd* is a handle to the desired window or control.

Return Value *nLength* is a short integer value specifying the text length. It is 0 if no such text exists.

FindWindow (*lpClassName*, *lpWindowName*) : *hWnd*

Purpose This function returns the handle of the window whose class is given by *lpClassName* and whose window name (caption) is given by *lpWindowName*.

Parameters *lpClassName* is a long pointer to a null-terminated string specifying the window's class name. If *lpClassName* is NULL, all class names match.

lpWindowName is a long pointer to a null-terminated string specifying the the window name (that is, the window's text caption). If *lpWindowName* is NULL, all window names match.

Return Value *hWnd* is the window handle of the window that has the specified class name and window name. It is NULL if no such window is found.

GetParent (*hWnd*) : *hWndParent*

Purpose This function retrieves the window handle of the specified window's parent (if any).

Parameter *hWnd* is a handle to the window whose parent window handle is to be retrieved.

Return Value *hWndParent* is a handle to the parent window. It is NULL if the window has no parent.

GetClientRect (*hWnd*, *lpRect*)

Purpose This function copies the client coordinates of a window's client area into the data structure pointed to by *lpRect*. The client coordinates specify the upper left and lower right corners of the client area. Since client coordinates are relative to the upper left corner of a window's client area, the coordinates of the upper left corner are 0,0.

Parameters *hWnd* is a handle to a window.

lpRect is a long pointer to a data structure having **RECT** type.

Return Value None.

GetWindowRect (*hWnd*, *lpRect*)

Purpose This function copies the dimensions of the bounding rectangle of the specified window into the structure pointed to by *lpRect*. The dimensions are given in screen coordinates, relative to the top left corner of the display screen, and include the caption, border, and scroll bars, if present.

Parameters *hWnd* is a handle to a window.
lpRect is a long pointer to a data structure of **RECT** type.

Return Value None.

GetSysModalWindow () : *hWnd*

Purpose This function returns the handle of a system modal window, if one is present.

Parameters None.

Return Value *hWnd* is a handle to the system modal window, if one is present. If no such window is present, *hWnd* is NULL.

SetSysModalWindow (*hWnd*) : *hPrevWnd*

Purpose This function makes the specified window a system modal window.

Parameter *hWnd* is a handle to the window to be made system modal.

Return Value *hPrevWnd* is a handle to the window that was previously the system modal window.

Notes If another window is made the active window (for example, the system modal window creates a dialog box that becomes the active window), the active window becomes the system modal window. When the original window becomes active again, it is system modal. To end the system modal state, destroy the window that is system modal.

2.17 Error Functions

This section describes the functions used to display errors and prompt for a user response.

MessageBox (*hWndParent*, *lpText*, *lpCaption*, *wType*) : *nMenuItem*

Purpose This function creates and displays a window that contains a application-supplied message and caption, plus any combination of the predefined icons and push buttons described below.

Parameters *hWndParent* is a handle to the window that owns the message box. The input focus is set to this window when **MessageBox** returns.

lpText is a long pointer to the message to be displayed. The string must be a null-terminated ASCII string.

lpCaption is a long pointer to the character string to be used for the dialog box caption. The string must be null-terminated. If *lpCaption* is NULL, the default caption "Error!" is used.

wType is an unsigned short integer value specifying the contents of the dialog box. It can be any combination of the following values, joined with the bitwise OR operator:

Value	Meaning
MB_OK	Message box contains an Ok push button.
MB_OKCANCEL	Message box contains Ok and Cancel push buttons.
MB_RETRYCANCEL	Message box contains Retry and Cancel push buttons.
MB_ABORTRETRYIGNORE	Message box contains three push buttons: Abort, Retry, and Ignore.
MB_YESNO	Message box contains two push buttons: Yes and No .

MB_YESNOCANCEL	Message box contains three push buttons: Yes, No, and Cancel.
MB_ICONHAND	A hand icon appears in the message box.
MB_ICONQUESTION	A question mark icon appears in the message box.
MB_ICONEXCLAMATION	An exclamation point icon appears in the message box.
MB_ICONASTERISK	An asterisk icon appears in the message box.
MB_DEFBUTTON1	First button is the default (the first button is always the default unless MB_DEFBUTTON2 or MB_DEFBUTTON3 is specified).
MB_DEFBUTTON2	Second button is the default.
MB_DEFBUTTON3	Third button is the default.
MB_APPLMODAL	The user must respond to the message box before continuing work in the window given by <i>hWndParent</i> . However, the user can move to the windows of other applications and work in those windows. (MB_APPLMODAL is the default.)
MB_SYSTEMMODAL	All applications are suspended until the user responds to the message box. System modal message boxes are used to notify the user of serious, potentially damaging, errors that require immediate attention (for example, running out of memory).

Return Value *nMenuItem* is one of the following menu item values returned by the dialog box:

IDOK
IDCANCEL
IDABORT
IDRETRY
IDIGNORE
IDYES
IDNO

If a message box has a “Cancel” button, the IDCANCEL value will be returned if either Escape or Cancel is pressed. If the message box has no “Cancel” button, pressing Escape has no effect.

Notes

When a system modal message box is created to indicate that the user is low on memory, the strings passed as the *lpText* and *lpCaption* parameters should not be taken from a resource file, since an attempt to load a resource file may fail. The message box can safely use the hand icon (MB_HANDICON), since this icon is always resident and does not require disk access.

When the keyboard interface is used to enumerate windows, the message box and its “parent” are considered to be “next” to each other.

If a message box is created while a dialog box is present, use the handle of the dialog box as the *hWndParent* parameter.

MessageBeep (*wType*) : *bBeep*

Purpose This function generates a “beep” at the system speaker when a message box is displayed.

Parameter *wType* is an unsigned short integer value that is the same as the *wType* parameter passed to the **MessageBox** function.

Return Value *bBeep*, a Boolean value, is nonzero if the function is successful, zero otherwise.

FlashWindow (*hWnd*, *bInvert*) : *bInverted*

Purpose

This function “flashes” the given window once. Flashing a window means changing the appearance of its caption bar or icon as if the window were changing from inactive to active status, or vice versa. (A gray caption bar or unframed icon changes to a black caption bar or framed icon; a black caption bar or framed icon changes to a gray caption bar or unframed icon.)

The window is flashed from one state to the other if the **bInvert** parameter is nonzero. If the *bInvert* parameter is zero, the window is returned to its original state (either active or inactive).

A window is typically flashed to indicate to the user that the window requires attention, but that it does not currently have the input focus.

Parameters

hWnd is a handle to a window that is either open or iconic.

bInvert, a Boolean value, is nonzero when the window is to be flashed. *bInvert* is zero if the window is to be returned to its original state (active or inactive).

Return Value

bInverted, a Boolean value, is nonzero if the window’s state is inverted. It is zero if an error occurs.

Notes

FlashWindow flashes the window only once; for successive flashing, the application should create a system timer.

bInvert should be zero only when the window is getting the input focus and will no longer be flashing; it should be nonzero on successive calls while waiting to get the input focus.

2.18 Cursor Functions

This section describes the functions used to set and move the mouse cursor. The mouse cursor is the primary pointing device used by Windows to provide input to an application.

SetCursor (*hCursor*) : *hOldCursor*

- Purpose* This function sets the cursor shape to the cursor specified by *hCursor*. The cursor is set only if the new shape is different from the existing shape. Otherwise, the function returns immediately. If *hCursor* is NULL, the cursor is removed from the screen.
- Parameters* *hCursor* is a handle to a cursor resource. The resource must have been loaded previously using the **LoadCursor** function.
- Return Value* *hOldCursor* is a handle to the cursor resource defining the previous cursor shape. It is NULL if there is no previous shape.
- Notes* This function is quite fast if *hCursor* is the same as the current cursor.
 The cursor is not shown on the screen if the cursor display count is less than zero (i.e., **HideCursor** has been called more times than **ShowCursor**.)

SetCursorPos (*X*, *Y*)

- Purpose* This function sets the position of the mouse cursor to the screen coordinates given by *X* and *Y*.
- Parameters* *X* and *Y* are integer values that specify the new screen coordinates of the mouse cursor.
- Return Value* None.
- Notes* To set the cursor to a position relative to a window, convert client coordinates to screen coordinates using the **ClientToScreen** function.

ClipCursor (*lpRect*)

- Purpose* This function is used to restrict the mouse cursor to a given rectangle on the screen.
- Parameter* *lpRect* is a long pointer to a data structure having **RECT** type. The cursor will be restricted to the rectangle defined by the screen coordinates given in the structure. If *lpRect* is NULL, the cursor can use the entire screen.
- Return Value* None.

GetCursorPos (*lpPoint*)

Purpose This function gets the current position, in screen coordinates, of the mouse cursor and stores it in the structure pointed to by *lpPoint*. The position returned is the true position when **GetCursorPos** was called; this function is NOT synchronized with the **GetMessage** and **PeekMessage** functions.

Parameter *lpPoint* is a long pointer to a data structure having **POINT** type.

Return Value None.

ShowCursor (*bShow*) : *nCount*

Purpose This function displays or hides the cursor. If *bShow* is nonzero, **ShowCursor** adds one to the display count. If *bShow* is zero, the display count is decreased by one.

The cursor is displayed only if the display count is greater than or equal to zero. Initially, the display count is zero if a mouse is installed, -1 otherwise.

Parameter *bShow* is a Boolean value that causes the display count to be increased if nonzero, and decreased if zero.

Return Value *nCount* is the new value of the display count.

Notes When a cursor is to be displayed in a mouseless environment, the application is responsible for defining its own user interface for moving the cursor.

2.19 Caret Functions

This section describes the functions used to create, display, move, and destroy the system caret. The system caret is a flashing bitmap that can be moved to any location on the display screen. The caret is typically used in text applications to mark a location.

CreateCaret (*hWnd*, *hBitmap*, *nWidth*, *nHeight*)

Purpose This function creates a caret for the given window. The caret has the shape and dimensions of the bitmap specified by *hBitmap*. If *hBitmap* is NULL, however, the caret is a solid flashing black block *nWidth* pixels wide by *nHeight* pixels high; if *hBitmap* is 1, the caret is gray instead of black.

The caret is initially hidden. The **ShowCaret** function must be called to display the caret.

Parameters *hWnd* is a handle to the window to receive the new caret. *hBitmap* is a handle to the bitmap defining the caret shape. If *hBitmap* is NULL, the caret is a solid black block; if *hBitmap* is 1, the caret is gray. *nWidth* and *nHeight* are integer values specifying the width and height of the caret. These values are used only if *hBitmap* is NULL or 1; otherwise, they are ignored.

Return Value None.

Notes If *nWidth* or *nHeight* is 0, then the system nominal border width or height is used. These values can be obtained by calling **GetSystemMetrics** with SM_CXBORDER and SM_CYBORDER as indexes. The width of a text caret is usually set to 0. This is preferable to giving 1 as the *nWidth* argument, since on a high resolution device a width of 1 may produce a very thin line.

Only one caret is available at a time, so it is not possible to have two carets flashing. An application should create a caret only when it has the keyboard focus or is the active window. Creating carets at other times may stop the caret from flashing in another application. Similarly, when an application becomes inactive or loses the focus, its caret should be destroyed.

DestroyCaret ()

Purpose This function destroys the current caret and frees any memory that it occupied.

Parameters None.

Return Value None.

Note An application must only call **DestroyCaret** if it has created a caret. Otherwise, the call to **DestroyCaret** may destroy the caret that is flashing in another application.

HideCaret (*hWnd*)

Purpose This function removes the system caret from the display screen, if the caret was created in the specified window. The caret is not destroyed; it can be redisplayed using the **ShowCaret** function.

Parameters *hWnd* is a handle to the window whose caret is to be hidden. If *hWnd* is NULL, the caret is hidden regardless of which window created it.

Return Value None.

ShowCaret (*hWnd*)

Purpose This function displays a caret created with the **CreateCaret** function, or redisplays a caret that has been removed from the display screen using the **HideCaret** function. The caret is displayed only if it was created in the specified window. If **HideCaret** has been called several times without any intervening calls to **ShowCaret**, an equal number of **ShowCaret** calls must be made before the caret is redisplayed.

Parameters *hWnd* is a handle to the window whose caret is to be redisplayed. If *hWnd* is NULL, the caret is redisplayed regardless of which window created it.

Return Value None.

SetCaretPos (*X*, *Y*)

Purpose This function moves the caret to the position specified by *X* and *Y*. The movement applies to the caret whether it is visible or hidden.

Parameters *X* and *Y* are integer values specifying the new position of the caret, in client coordinates of the window for which the caret was created.

Return Value None.

Note The application is responsible for ensuring that **SetCaretPos** is called only when the caret is in the application's own window. If **SetCaretPos** is called when the caret is in another window, the caret position will be changed.

SetCaretBlinkTime (*nMSeconds*)

Purpose This function establishes the caret flash rate.

Parameters *nMSeconds* is an integer value that gives the delay, in milliseconds.

Return Value None.

GetCaretBlinkTime () : *nMSeconds*

Purpose This function returns the current caret flash rate.

Parameters None.

Return Value *nMSeconds* gives the delay, in milliseconds.

2.20 Coordinate Functions

This section describes the functions used to convert client coordinates to screen coordinates, and vice versa. Client coordinates refer to points within the client area of a window and are always relative to the upper left corner of the client area. Screen coordinates refer to points on the screen and are relative to the upper left corner of the screen.

ClientToScreen (*hWnd*, *lpPoint*)

Purpose This function converts the client coordinates of the point specified by *lpPoint* into equivalent screen coordinates. Screen coordinates are relative to the upper left corner of the screen and do not refer to any specific window.

ClientToScreen assumes that the point to be converted lies inside the client area of the specified window. The function copies the new screen coordinates into the location pointed to by *lpPoint*, overwriting the original client coordinates.

Parameters *hWnd* is a handle to a window.

lpPoint is a long pointer to a data structure having **POINT** type.

Return Value None.

ScreenToClient (*hWnd*, *lpPoint*)

Purpose This function converts the screen coordinates of the point specified by *lpPoint* into equivalent client coordinates. The point is assumed to be in screen coordinates. The new client coordinates replace the existing values in *lpPoint*.

Parameters *hWnd* is a handle to the window whose client area is used in the mapping.

lpPoint is a long pointer to a data structure having **POINT** type.

Return Value None.

WindowFromPoint (*Point*) : *hWnd*

Purpose This function identifies the window that contains the given point. The *Point* parameter must specify the screen coordinates of a point on the screen.

Parameters *Point* is a data structure of **POINT** type.

Return Value *hWnd* is the handle of the window in which the point lies. It is NULL if no window exists at the given point.

ChildWindowFromPoint (*hWndParent*, *Point*) : *hWndChild*

Purpose This function determines which, if any, of the child windows belonging to the given parent window contains the specified point.

Parameters *hWndParent* is a handle to the parent window.

Point is a data structure with **POINT** type containing the client coordinates of the point to be tested.

Return Value *hWndChild* is a handle to the child window containing the point. It is NULL if the given point lies outside the parent window. If the point is within the parent window but is not contained within any child window, the handle of the parent window is returned.

2.21 Rectangle Functions

This section describes the utility functions used to create and modify rectangle data structures. Rectangle data structures are required as input by many GDI functions.

SetRect (*lpRect, X1, Y1, X2, Y2*)

Purpose This function fills the structure pointed to by *lpRect* with the coordinates given by *X1*, *Y1*, *X2*, and *Y2*.

Parameters *lpRect* is a long pointer to a data structure having **RECT** type.

X1 and *Y1* are short integer values specifying the coordinates of the upper left corner of the rectangle.

X2 and *Y2* are short integer values specifying the coordinates of the lower right corner of the rectangle.

Return Value None.

Notes The width of the rectangle must not exceed 32K units.

SetRectEmpty (*lpRect*)

Purpose This function sets the rectangle to an empty rectangle (all coordinates zero).

Parameters *lpRect* is a long pointer to a data structure having **RECT** type.

Return Value None.

CopyRect (*lpDestRect, lpSourceRect*)

Purpose This function makes a copy of an existing rectangle.

Parameters *lpDestRect* is a long pointer to a destination data structure having **RECT** type.

lpSourceRect is a long pointer to a source data structure having **RECT** type.

Return Value None.

InflateRect (*lpRect*, *X*, *Y*)

Purpose

This function expands the rectangle specified by *lpRect*, increasing the width by *X* units on the left and right ends of the rectangle, and increasing the height by *Y* units on the top and bottom. If *X* or *Y* is negative, the width or height of the rectangle is decreased.

Parameters

lpRect is a long pointer to a data structure having **RECT** type.

X is a short integer value specifying the amount to increase the rectangle width. If *X* is negative, the width is decreased.

Y is a short integer value specifying the amount to increase the rectangle height. If *Y* is negative, the height is decreased.

Return Value

None.

Notes

The width of the rectangle must not exceed 32K units.

IntersectRect (*lpDestRect*, *lpSrc1Rect*, *lpSrc2Rect*) : *nIntersection*

Purpose

This function finds the intersection of two rectangles and copies it to the buffer specified by *lpDestRect*.

Parameters

lpDestRect is a long pointer to the buffer to receive the intersection. It must have **RECT** type.

lpSrc1Rect is a long pointer to a data structure having **RECT** type.

lpSrc2Rect is a long pointer to a data structure having **RECT** type.

Return Value

nIntersection, an integer value, is nonzero if the intersection of the two rectangles is not empty. *nIntersection* is zero if the intersection is empty.

UnionRect (*lpDestRect*, *lpSrc1Rect*, *lpSrc2Rect*) : *nUnion*

Purpose

This function creates the union of two rectangles. The union is equal to the smallest rectangle containing both source rectangles.

Parameters

lpDestRect is a long pointer to a data structure having **RECT** type.

lpSrc1Rect is a long pointer to a data structure having **RECT** type.

lpSrc2Rect is a long pointer to a data structure having **RECT** type.

Return Value *nUnion*, an integer value, is 1 if the union is not empty. It is 0 if the union is empty.

OffsetRect (*lpRect*, *X*, *Y*)

Purpose This function moves the specified rectangle by the specified offsets. **OffsetRect** moves the rectangle *X* units along the x-axis and *Y* units along the y-axis.

Parameters *lpRect* is a long pointer to a data structure having **RECT** type.

X is a short integer value specifying the amount to move left or right.

Y is a short integer value specifying the amount to move up or down.

Return Value None.

IsRectEmpty (*lpRect*) : *bEmpty*

Purpose This function determines whether or not the specified rectangle is empty. A rectangle is empty if the width or height is zero.

Parameters *lpRect* is a long pointer to a data structure having **RECT** type.

Return Value *bEmpty*, a Boolean value, is nonzero if the rectangle is empty. Otherwise, it is zero.

PtInRect (*lpRect*, *Point*) : *bInRect*

Purpose This function indicates whether or not a specified point lies within a given rectangle.

Parameters *lpRect* is a long pointer to the rectangle to be checked. It must point to a data structure having **RECT** type.

Point is a data structure having **POINT** type.

Return Value *bInRect*, a Boolean value, is nonzero if the point lies within the given rectangle. Otherwise, it is zero.

2.22 System Information Functions

This section describes functions used to retrieve and change system characteristics.

GetSystemMetrics (*nIndex*) : *nValue*

Purpose This function retrieves information about system metrics.

Parameters *nIndex* is one of the following integer values, specifying which measurement is to be retrieved:

Index	Value
SM_CXSCREEN	Width of screen
SM_CYSCREEN	Height of screen
SM_CXVSCROLL	Width of arrow bitmap on vertical scroll bar
SM_CYVSCROLL	Height of arrow bitmap on vertical scroll bar
SM_CXHSCROLL	Width of arrow bitmap on horizontal scroll bar
SM_CYHSCROLL	Height of arrow bitmap on horizontal scroll bar
SM_CYCAPTION	Height of caption
SM_CXBORDER	Width of left/right window border
SM_CYBORDER	Height of top/bottom window border
SM_CXDLGFRAME	Width of frame when window has WS_DLGFREAME style
SM_CYDLGFRAME	Height of frame when window has WS_DLGFREAME style
SM_CYVTHUMB	Height of thumb box on vertical scroll bar

SM_CXHTHUMB	Width of thumb box on horizontal scroll bar
SM_CXICON	Width of icon
SM_CYICON	Height of icon
SM_CXCURSOR	Width of cursor
SM_CYCURSOR	Height of cursor
SM_CYMENU	Height of single line menu bar
SM_CXFULLSCREEN	Width of window client area for full screen window
SM_CYFULLSCREEN	Height of window client area for full screen window (equivalent to the height of the screen minus the height of the window caption)
SM_CYKANJIWINDOW	Height of Kanji window
SM_MOUSEPRESENT	Non-zero if mouse hardware installed
SM_DEBUG	Non-zero if windows debugging version
SM_CMETRICS	Number of system metrics
SM_FULLSCREEN	Handle to full-screen window; zero if no such window is present
SM_CURSORLEVEL	Cursor display count (see the ShowCursor function).

Return Value *nValue* is an integer measurement whose meaning depends on the *nIndex* parameter.

GetSysColor (*nIndex*) : *rgbColor*

Purpose This function retrieves information about system colors.

Parameter *nIndex* is one of the following integer values, specifying the item whose color is to be retrieved:

Index	Item
COLOR_SCROLLBAR	Scroll bar "gray" area
COLOR_BACKGROUND	Desktop
COLOR_ACTIVECAPTION	Active window caption
COLOR_INACTIVECAPTION	Inactive window caption
COLOR_MENU	Menu background
COLOR_WINDOW	Window background and thumb box
COLOR_WINDOWFRAME	Window border, caption text background
COLOR_MENUTEXT	Text in menus
COLOR_WINDOWTEXT	Text in windows
COLOR_CAPTIONTEXT	Text in caption, size box, scroll bar arrow box

Return Value *rgbColor* is an RGB color value specifying the color of the specified item.

SetSysColors (*nChanges*, *lpSysColor*, *lpColorValues*)

Purpose This function changes one or more system colors.

Parameters *nChanges* is an integer specifying the number of colors to be changed.

lpSysColor is a long pointer to an array of integer indexes specifying the items to be changed. The array consists of one or more of the index values described above for **GetSysColor**.

lpColorValues is a long pointer to an array of unsigned long integers that represent RGB color values.

Return Value None.

Note **SetSysColors** does NOT write the color change to the WIN.INI file.

2.23 Window Hook Function

The window hook function is a general-purpose function for installing one of several kinds of "hooks" into Windows. When a particular type of application-supplied hook function is installed, Windows calls the hook function before carrying the normal Windows processing for that type of input or action. This allows the application to intercept and alter the normal sequence of Windows processing.

SetWindowsHook (*nFilterType*, *lpFilterFunc*) : *lpPrevFilterFunc*

Purpose This function is used to install system and/or application hook functions. The purpose of each hook is explained below.

Parameters *nFilterType* is one of the following short integer values, indicating which hook is being installed:

Value	Meaning
WH_MSGFILTER	Install a message filter. The filter function pointed to by <i>lpFilterFunc</i> will be called whenever a message occurs, before Windows performs its default processing of the message.
WH_KEYBOARD	Install a keyboard filter. The filter function pointed to by <i>lpFilterFunc</i> will be called whenever a WM_KEYUP or WM_KEYDOWN message occurs; the message is passed to the filter function before being sent to the window with the input focus.

lpFilterFunc is a long pointer to the filter function to be installed. The function is passed a pointer to a **MSG** structure and a code that indicates the Windows message-processing context. The code is an integer with one of the following values:

Code	Meaning
MSGF_DIALOGBOX	Processing messages inside DialogBox
MSGF_MESSAGEBOX	Processing messages inside MessageBox
MSGF_MENU	Menu tracking

The filter function can handle the message in one of three ways:

1. Process the message and return nonzero. Windows will do no further processing of the message.
2. Return zero without processing the message. Windows will perform the usual processing.
3. Modify the message and return zero. Windows will perform the usual processing on the modified message.

Return Value *lpPrevFilterFunc* is the address of the previously installed filter, if any. If the application chooses not to process a given message, it should call the previously installed filter and pass it the message information. This allows a chain of filter functions to be created.

Notes The address passed as the *lpFilterFunc* parameter must be created using **MakeProcInstance**.

The filter function must use the Pascal calling convention and must be declared FAR. The number and type of the parameters depends on whether a STACKSIZE statement appears in the .DEF file for the application. For applications that use a STACKSIZE statement (SS==DS), *lpFilterFunc* must be declared as follows:

FilterFunc(wDummy, pMsg, nCode) : bProcess

For applications that do not use a STACKSIZE statement (SS!=DS), *lpFilterFunc* must be declared as follows:

FilterFunc(lpMsg, nCode) : bProcess

Chapter 3

GDI Functions

3.1	Introduction	107
3.2	Display Context Functions	107
3.3	Output Functions	110
3.4	Drawing Object Functions	127
3.5	Selection Functions	138
3.6	Display Context Attribute Functions	140
3.7	Clipping Region Functions	159
3.8	Region Functions	162
3.9	Text Justification Functions	165
3.10	Metafile Functions	168
3.11	Control Functions	171
3.12	GDI Information Functions	180
3.13	Conversion Functions	187

(

(

(

3.1 Introduction

This chapter describes the Graphics Device Interface (GDI) functions. These functions make up a special graphics package that provides access to a large variety of graphic and other display devices.

3.2 Display Context Functions

This section describes the functions used to create and destroy display contexts for graphics devices. A display context is a set of data, maintained by GDI, that describes a graphics device and its device driver.

CreateDC (*lpDriverName*, *lpDeviceName*, *lpOutput*, *lpInitData*) : *hDC*

Purpose This function creates a display context for the specified device. The *lpDriverName*, *lpDeviceName*, and *lpOutput* parameters specify the device driver, device name, and physical output medium (file or port).

Parameters *lpDriverName* is a long pointer to a null-terminated ASCII string specifying the MS-DOS filename, without extension, of the device driver (for example, "EPSON").

lpDeviceName is a long pointer to a null-terminated ASCII string specifying the name of the specific device to be supported (for example, "EPSON FX-80"). This parameter is used if the module supports more than one device.

lpOutput is a long pointer to a null-terminated ASCII string specifying the MS-DOS file or device name for the physical output medium (file or output port).

lpInitData is a long pointer to device-specific initialization data for the device driver. *lpInitData* is NULL if the device driver requires no initialization.

Return Value If the function is successful, *hDC* is a handle to a display context for the specified device. Otherwise, *hDC* is NULL.

Notes MS-DOS device names follow MS-DOS conventions: an ending colon is recommended, but optional. Windows strips the

terminating colon so that a device name ending with a colon is mapped to the same port as the same name without a colon.

The driver and port names must not contain leading or trailing spaces.

CreateCompatibleDC (*hDC*) : *hMemDC*

Purpose

This function creates a memory display context that is compatible with the device specified by *hDC*. A memory display context is a block of memory that represents a display surface. The memory display can be used to prepare images in memory before copying them to the actual display surface of the compatible device.

When a memory display context is created, GDI automatically selects a monochrome stock bitmap for it.

Parameters

hDC is a handle to a display context. If *hDC* is NULL, the function creates a memory display context that is compatible with the system display.

Return Value

If the function is successful, *hMemDC* is a handle to a memory display context. Otherwise, it is NULL.

Notes

This function can only be used to create compatible display contexts for devices that support raster operations. See the **GetDeviceCaps** function and the RC_BITBLT capability.

GDI output functions can be used with a memory display context only if a bitmap has been created and selected into that context.

CreateIC (*lpDriverName*, *lpDeviceName*, *lpOutput*, *lpInitData*) : *hIC*

Purpose

This function creates an information context for the specified device. The information context provides a fast way to get information about the device without creating a display context.

Parameters

lpDriverName is a long pointer to a null-terminated ASCII string specifying the MS-DOS filename, without extension, of the device driver (for example, "EPSON").

lpDeviceName is a long pointer to a null-terminated ASCII string specifying the name of the specific device to be supported (for example, "EPSON FX-80"). This parameter is used if the module supports more than one device.

lpOutput is a long pointer to a null-terminated ASCII string specifying the MS-DOS file or device name for the physical output medium (file or port).

lpInitData is a long pointer to device-specific initialization data for the device driver. *lpInitData* is NULL if the device driver is to use the default initialization (if any) specified by the user through the control panel.

Return Value If the function is successful, *hIC* is a handle to an information context for the specified device. Otherwise, it is NULL.

Notes MS-DOS device names follow MS-DOS conventions: an ending colon is recommended, but optional. Windows strips the terminating colon so that a device name ending with a colon is mapped to the same port as the same name without a colon.

The driver and port names must not contain leading or trailing spaces.

GDI output functions cannot be used with information contexts.

DeleteDC (*hDC*) : *bDeleted*

Purpose This function deletes the specified display context. If *hDC* is the last display context for a given device, the device is notified and all storage and system resources used by the device are released.

Parameters *hDC* is a handle to a display context.

Return Value *bDeleted*, a Boolean value, is nonzero if the display context is successfully deleted (regardless of whether the deleted display context is the last display context for the device). If an error occurs, *bDeleted* is zero.

SaveDC (*hDC*) : *nSavedDC*

Purpose This function saves the current state of the display context *hDC* by copying state information (such as clipping region, selected objects, and mapping mode) to a context stack. The saved display context can later be restored by using the **RestoreDC** function.

Parameters *hDC* is a handle to the display context to be saved.

Return Value $nSavedDC$ is a short integer value uniquely identifying the saved display context. It is 0 if an error occurs.

Notes **SaveDC** can be used any number of times to save any number of display context states.

RestoreDC ($hDC, nSavedDC$) : $bRestored$

Purpose This function restores the display context specified by hDC to the previous state identified by $nSavedDC$. **RestoreDC** restores the context by copying state information saved on the context stack by earlier calls to **SaveDC**.

The context stack can contain the state information for several display contexts. If the context specified by $nSavedDC$ is not at the top of the stack, **RestoreDC** deletes any state information between $nSavedDC$ and the top of the stack. The deleted information is lost.

Parameters hDC is a handle to a display context.

$nSavedDC$ is a short integer value specifying the device context to be restored. It can be a value returned by a previous **SaveDC** call. If $nSavedDC$ is -1, the most recent display context saved is restored.

Return Value $bRestored$, a Boolean value, is nonzero if the specified context is restored. Otherwise, it is zero.

3.3 Output Functions

This section describes the functions used to create images on a display surface.

MoveTo (hDC, X, Y) : $ptPrevPos$

Purpose This function moves the current position to the point specified by X and Y .

Parameters hDC is a handle to a display context.

X and Y are short integer values specifying the logical coordinates of the new position.

Return Value *ptPrevPos* is a long integer specifying the x and y coordinates of the previous position. The y coordinate is in the high-order word; the x coordinate is in the low-order word.

Note Although **MoveTo** has no output, it affects other output functions that use the current position.

GetCurrentPosition (*hDC*) : *ptPos*

Purpose This function retrieves the logical coordinates of the current position.

Parameters *hDC* is a handle to a display context.

Return Value *ptPos* is a long unsigned integer specifying the current position. The y coordinate is in the high-order word; the x coordinate is in the low-order word.

LineTo (*hDC*, *X*, *Y*) : *bDrawn*

Purpose This function draws a line from the current position up to, but not including, the point specified by *X* and *Y*. The line is drawn with the currently selected pen. If no error occurs, the current position is set to (*X*, *Y*).

Parameters *hDC* is a handle to a display context.

X and *Y* are short integer values specifying the logical coordinates of the end point for the line.

Return Value *bDrawn*, a Boolean value, is nonzero if the line is drawn. Otherwise, it is zero.

Polyline (*hDC*, *lpPoints*, *nCount*) : *bDrawn*

Purpose This function draws a set of line segments, connecting the points specified by the *lpPoints* parameter. The lines are drawn from the first point through subsequent points with a result as if the **MoveTo** and **LineTo** functions were used to move to each new point and connect it to the next. (However, the current position is neither used nor updated by **Polyline**).

Parameters *hDC* is a handle to a display context.

lpPoints is a long pointer to an array of points to be connected. Each element in the array must have **POINT** type. The x and y coordinates of the points are assumed to be logical coordinates.

nCount is a short integer value specifying the number of points in the array. It must be at least 2.

Return Value *bDrawn*, a Boolean value, is nonzero if the line segments are drawn. Otherwise, *bDrawn* is zero.

Rectangle (*hDC, X1, Y1, X2, Y2*) : *bDrawn*

Purpose This function draws a rectangle. The interior of the rectangle is filled using the currently selected brush, and a border is drawn with the currently selected pen.

Parameters *hDC* is a handle to a display context.

X1 and *Y1* are short integer values specifying the logical coordinates of the upper left corner of the rectangle.

X2 and *Y2* are short integer values specifying the logical coordinates of the lower right corner of the rectangle.

Return Value *bDrawn*, a Boolean value, is nonzero if the rectangle is drawn. Otherwise, *bDrawn* is zero.

Notes The width of the rectangle specified by *X1*, *Y1*, *X2*, and *Y2* must not exceed 32,767 units.

The current position is neither used nor updated by this function.

RoundRect (*hDC, X1, Y1, X2, Y2, X3, Y3*) : *bDrawn*

Purpose This function draws a rectangle with rounded corners. The interior of the rectangle is filled using the currently selected brush, and a border is drawn with the currently selected pen.

Parameters *hDC* is a handle to a display context.

X1 and *Y1* are short integer values specifying the logical coordinates of the upper left corner of the rectangle.

X2 and *Y2* are short integer values specifying the logical coordinates of the lower right corner of the rectangle.

X3 and *Y3* are short integer values specifying the width and height of the ellipse to be used to draw the rounded corners.

Return Value *bDrawn*, a Boolean value, is nonzero if the rectangle is drawn. Otherwise, *bDrawn* is zero.

Notes The width of the rectangle specified by *X1*, *Y1*, *X2*, and *Y2* must not exceed 32,767 units.

The current position is neither used nor updated by this function.

Polygon (*hDC*, *lpPoints*, *nCount*) : *bDrawn*

Purpose This function draws a polygon consisting of two or more points (vertices) connected by lines. The lines are drawn according to the current polygon filling mode. In ALTER-NATE mode, lines are drawn from the first point through subsequent points using the current pen, and the interior is filled with the current brush. In WINDING mode, all points are used to compute a border, the border is then drawn using the current pen, and the interior filled with the current brush. (See the **SetPolyFillMode** for a description of how WINDING mode borders are computed.) In both modes, the polygon is automatically closed, if necessary, by drawing a line from the last vertex to the first.

Parameters *hDC* is a handle to a display context.

lpPoints is a long pointer to an array of points specifying the vertices of the polygon. Each point in the array must have **POINT** type. The x and y coordinates are assumed to be logical coordinates.

nCount is a short integer value specifying the number of vertices given in the *lpPoints* array.

Return Value *bDrawn*, a Boolean value, is nonzero if the function is successful. Otherwise, it is zero.

Notes The current position is neither used nor updated by this function.

The current polygon filling mode can be retrieved or set using the **GetPolyFillMode** and **SetPolyFillMode** functions.

Ellipse (*hDC*, *X1*, *Y1*, *X2*, *Y2*) : *bDrawn*

Purpose This function draws an ellipse. The center of the ellipse is the center of the bounding rectangle specified by *X1*, *Y1*, *X2*, and *Y2*. The ellipse border is drawn with the current pen, and the interior is filled with the current brush.

If the bounding rectangle is empty, nothing is drawn.

<i>Parameters</i>	<i>hDC</i> is a handle to a display context.
	<i>X1</i> and <i>Y1</i> are short integer values specifying the logical coordinates of the upper left corner of the bounding rectangle.
	<i>X2</i> and <i>Y2</i> are short integer values specifying the logical coordinates of the lower right corner of the bounding rectangle.
<i>Return Value</i>	<i>bDrawn</i> , a Boolean value, is nonzero if the ellipse is drawn. Otherwise, <i>bDrawn</i> is zero.

Notes The width of the rectangle specified by *X1*, *Y1*, *X2*, and *Y2* must not exceed 32K units.

The current position is neither used nor updated by this function.

Arc (*hDC*, *X1*, *Y1*, *X2*, *Y2*, *X3*, *Y3*, *X4*, *Y4*) : *bDrawn*

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

Parameters *hDC* is a handle to a display context.

X1 and *Y1* are short integer values specifying the logical coordinates of the upper left corner of the bounding rectangle.

X2 and *Y2* are short integer values specifying the logical coordinates of the lower right corner of the bounding rectangle.

X3 and *Y3* are short integer values specifying the logical coordinates of the arc's starting point. This point does not have to lie exactly on the arc.

X4 and *Y4* are short integer values specifying the logical coordinates of the arc's endpoint. This point does not have to lie exactly on the arc.

Return Value *bDrawn*, a Boolean value, is nonzero if the arc is drawn. Otherwise, it is zero.

Notes The width of the rectangle specified by $X1$, $Y1$, $X2$, and $Y2$ must not exceed 32,767 units.

Pie (*hDC*, *X1*, *Y1*, *X2*, *Y2*, *X3*, *Y3*, *X4*, *Y4*) : *bDrawn*

Purpose This function draws an pie-shaped wedge by drawing an elliptical arc whose center and two endpoints are joined by lines. The center of the arc is the center of the bounding rectangle specified by $X1$, $Y1$, $X2$, and $Y2$. The starting and ending points of the arc are specified by $X3$, $Y3$, $X4$, and $Y4$. The arc is drawn with the currently selected pen moving in a counterclockwise direction. Two additional lines are drawn from each endpoint to the arc's center. The pie-shaped area is filled with the currently selected brush.

Parameters hDC is a handle to a display context.

$X1$ and $Y1$ are short integer values specifying the logical coordinates of the upper left corner of the bounding rectangle.

$X2$ and $Y2$ are short integer values specifying the logical coordinates of the lower right corner of the bounding rectangle.

$X3$ and $Y3$ are short integer values specifying the logical coordinates of the starting point of the arc. This point does not have to lie exactly on the arc.

$X4$ and $Y4$ are short integer values specifying the logical coordinates of the endpoint of the arc. This point does not have to lie exactly on the arc.

Return Value $bDrawn$, a Boolean value, is nonzero if the pie shape is drawn. Otherwise, it is zero.

Notes The width of the rectangle specified by $X1$, $Y1$, $X2$, and $Y2$ must not exceed 32,767 units.

The current position is neither used nor updated by this function.

PatBlt (*hDC*, *X*, *Y*, *nWidth*, *nHeight*, *dwRop*) : *bDrawn*

Purpose This function creates a bit pattern on the specified device. The pattern is a combination of the currently selected brush and the pattern already on the device. The raster operation code, $dwRop$, defines how the patterns are to be combined.

<i>Parameters</i>	<i>hDC</i> is a handle to a display context. <i>X</i> and <i>Y</i> are short integer values specifying the logical coordinates of the upper left corner of the rectangle to receive the pattern. <i>nWidth</i> and <i>nHeight</i> are short integer values specifying the width and height (in logical units) of the rectangle to receive the pattern. <i>dwRop</i> is a long unsigned integer specifying the raster operation code.
<i>Return Value</i>	<i>bDrawn</i> , a Boolean value, is nonzero if the bit pattern is drawn. Otherwise, <i>bDrawn</i> is zero.
<i>Note</i>	<i>dwRop</i> values for this function are a limited subset of the full 256 ternary raster operation codes; in particular, no operation code that refers to a source can be used. Not all devices support the PatBlt function. See the GetDeviceCaps function and the RC_BITBLT capability.
BitBlt (<i>hDestDC</i>, <i>X</i>, <i>Y</i>, <i>nWidth</i>, <i>nHeight</i>, <i>hSrcDC</i>, <i>XSrc</i>, <i>YSrc</i>, <i>dwRop</i>) : <i>bDrawn</i>	
<i>Purpose</i>	This function moves a bitmap from the source device given by <i>hSrcDC</i> to the destination device given by <i>hDestDC</i> . The <i>XSrc</i> and <i>YSrc</i> parameters specify the origin on the source device of the bitmap to be moved. The <i>X</i> , <i>Y</i> , <i>nWidth</i> , and <i>nHeight</i> parameters specify the origin, width, and height of the rectangle on the destination device to be filled by the bitmap. The raster operation, <i>dwRop</i> , defines how the bits of the source and destination are combined.
<i>Parameters</i>	<i>hDestDC</i> is a handle to a display context for the destination device. <i>X</i> and <i>Y</i> are short integer values specifying the logical coordinates of the upper left corner of the destination rectangle. <i>nWidth</i> and <i>nHeight</i> are short integer values specifying the width and height in logical units of the destination rectangle and source bitmap. <i>hSrcDC</i> is a handle to a display context for the source device. It must be NULL if <i>dwRop</i> specifies a raster operation that does not include a source. <i>XSrc</i> and <i>YSrc</i> are short integer values specifying the logical coordinates of the upper left corner of the source bitmap.

dwRop is a long unsigned integer value specifying the raster operation to be performed.

Return Value *bDrawn*, a Boolean value, is nonzero if the bitmap is drawn. Otherwise, *bDrawn* is zero.

Notes GDI transforms the *nWidth* and *nHeight* once using the destination display context, and once using the source display context. If the resulting extents do not match, GDI uses **StretchBlt** to compress or stretch the source bitmap as necessary.

If destination, source, and pattern bitmaps do not have the same color format, **BitBlt** converts the source and pattern bitmaps to match the destination. The foreground and background colors of the destination are used in the conversion.

If **BitBlt** converts monochrome bitmaps to color, it sets white bits (1) to background color and black bits (0) to foreground color. To convert color to monochrome, the function sets pixels that match the background color to white (1), and all other pixels to black (0). The foreground and background colors of the display context with color are used.

Not all devices support the **BitBlt** function. See the **GetDeviceCaps** function and the RC_BITBLT capability.

StretchBlt (*hDestDC, X, Y, nWidth, nHeight, hSrcDC, XSrc, YSrc, nSrcWidth, nSrcHeight, dwRop*) : *bDrawn*

Purpose This function moves a bitmap from a source rectangle into a destination rectangle, stretching or compressing the bitmap, if necessary, to fit the dimensions of the destination rectangle. **StretchBlt** uses the stretching mode of the destination display context (set by **SetStretchBltMode**) to determine how to stretch or compress the bitmap.

StretchBlt moves the bitmap from the source device given by *hSrcDC* to the destination device given by *hDestDC*. The *XSrc*, *YSrc*, *nSrcWidth*, and *nSrcHeight* parameter define the origin and dimensions of the source rectangle. The *X*, *Y*, *nWidth*, and *nHeight* parameters give the origin and dimensions of the destination rectangle. The raster operation, *dwRop*, defines how the source bitmap and the bits already on the destination device are combined.

StretchBlt creates a mirror image of a bitmap if the signs of the *nSrcWidth* and *nWidth* or *nSrcHeight* and *nHeight*

differ. If *nSrcWidth* and *nWidth* have different signs, the function creates a mirror image of the bitmap along the x-axis. If *nSrcHeight* and *nHeight* have different signs, the function creates a mirror image of the bitmap along the y-axis.

Parameters *hDestDC* is a handle to a display context for the destination device.

X and *Y* are short integer values specifying the logical coordinates of the upper left corner of the destination rectangle.

nWidth and *nHeight* are short integer values specifying the width and height in logical units of the destination rectangle.

hSrcDC is a handle to a display context for the source device.

XSrc and *YSrc* are short integer values specifying the logical coordinates of the upper left corner of the source rectangle.

nSrcWidth and *nSrcHeight* are short integer values specifying the width and height in logical units of the source rectangle.

dwRop is a long unsigned integer specifying the raster operation code.

Return Value *bDrawn*, a Boolean value, is nonzero if the bitmap is drawn. Otherwise, *bDrawn* is zero.

Notes **StretchBlt** stretches or compresses the source bitmap in memory, then copies the result to the destination. If a pattern is to be merged with the result, it is not merged until the stretched source bitmap is copied to the destination.

If a brush is used, it is the currently selected brush in the destination display context.

The destination coordinates are transformed according to the destination display context; the source coordinates are transformed using the source display context.

If destination, source, and pattern bitmaps do not have the same color format, **StretchBlt** converts the source and pattern bitmaps to match the the destination. The foreground and background colors of the destination are used in the conversion.

If **StretchBlt** must convert a monochrome bitmap to color, it sets white bits (1) to background color and black bits (0)

to foreground color. To convert color to monochrome, it sets pixels that match the background color to white (1), and all other pixels to black (0). The foreground and background colors of the color display context are used.

Not all devices support the **StretchBlt** function. See the **GetDeviceCaps** function and the RC_BITBLT capability.

TextOut (*hDC*, *X*, *Y*, *lpString*, *nCount*) : *bDrawn*

<i>Purpose</i>	This function writes a character string on the specified display, using the currently selected font. The starting position of the string is given by the <i>X</i> and <i>Y</i> parameters.
<i>Parameters</i>	<p><i>hDC</i> is a handle to a display context.</p> <p><i>X</i> and <i>Y</i> are short integer values specifying the logical coordinates of the starting point of the string.</p> <p><i>lpString</i> is a long pointer to the character string to be drawn.</p> <p><i>nCount</i> is a short integer value specifying the number of characters in the string.</p>
<i>Return Value</i>	<i>bDrawn</i> , a Boolean value, is nonzero if the string is drawn. Otherwise, <i>bDrawn</i> is zero.
<i>Notes</i>	Character origins are defined to be at the upper left corner of the character cell. The current position is not used or updated by this function.

DrawText (*hDC*, *lpString*, *nCount*, *lpRect*, *wFormat*)

<i>Purpose</i>	This function draws formatted text in the rectangle specified by <i>lpRect</i> . The text is drawn using the currently selected text color and background color in the display context given by <i>hDC</i> . The output is clipped to the rectangle specified by <i>lpRect</i> (unless the DT_NOCLIP format is used).
	A carriage return character returns the pen to the left edge of the rectangle, and a line feed character moves the pen down one line (unless the DT_SINGLELINE format is used). Carriage return/line feed sequences cause the line to wrap.
<i>Parameters</i>	<p><i>hDC</i> is a handle to a display context.</p> <p><i>lpString</i> is a long pointer to the character string to be drawn.</p>

nCount is a short integer value specifying the number of bytes in the string. If *nCount* is -1, then *lpString* is assumed to be a long pointer to a null-terminated string.

lpRect is a long pointer to a data structure having **RECT** type. The rectangle structure specifies the clipping rectangle in logical coordinates.

wFormat is an integer value specifying the method of formatting the text. It can be any combination of the following:

Value	Meaning
DT_LEFT	Left justified text.
DT_CENTER	Centered text.
DT_RIGHT	Right justified text.
DT_SINGLELINE	Single line only. Carriage returns and line feeds do not break the line.
DT_TOP	Top justified (single line only).
DT_VCENTER	Vertically centered (single line only).
DT_BOTTOM	Bottom justified (single line only).
DT_WORDBREAK	Word breaking. Lines are automatically broken between words if a word would extend past the edge of the rectangle specified by <i>lpRect</i> . Carriage return/line feed sequences also cause a line break.
DT_EXPANDTABS	Expand tab characters (default number of characters per tab is 8).
DT_TABSTOP	Set tab stops. The high-order byte of <i>wFormat</i> is the number of characters for each tab. Default is 8.

DT_NOCLIP Draw without clipping. **DrawText** is somewhat faster when **DT_NOCLIP** is used.

DT_EXTERNALLEADING

Include the font external leading in line height. Normally, external leading is not included in the height of a line of text.

The values can be combined with the bitwise OR operator.

Return Value None.

Note If the current font is too large for the specified rectangle, the function will not attempt to substitute a smaller font.

GrayString (*hDC, hBrush, lpOutputFunc, lpData, nCount, X, Y, nWidth, nHeight*) : *bDrawn*

Purpose This function writes and grays a character string. The string is grayed regardless of the currently selected brush and background.

Parameters *hDC* is a handle to a display context.

hBrush is a handle to the brush to be used for graying.

lpOutputFunc is a long pointer to a function to draw the string. If *lpOutputFunc* is NULL, GDI uses the **TextOut** function, and *lpData* is assumed to be a long pointer to the character string to be output. If the characters to be output cannot be handled by **TextOut** (for example, the string is stored as a bitmap), the application must supply its own output function.

lpData is a long pointer to data to be passed to the output function. If *lpOutputFunc* is NULL, *lpData* must be a long pointer to the string to be output.

nCount is a short integer specifying the number of characters to be output. If *nCount* is zero, **GrayString** calculates the length of the string (assuming that *lpData* is a pointer to the string). If *nCount* is -1 and the function pointed to by *lpOutputFunc* returns zero, the image is shown but not grayed.

X and *Y* are short integer values specifying the x and y coordinates (in logical units) of the starting position of the rectangle enclosing the string.

nWidth and *nHeight* are short integer values specifying the width and height (in logical units) of the rectangle enclosing the string. If either *nWidth* or *nHeight* is zero, **GrayString** calculates the dimensions of the area (assuming *lpData* is a pointer to the string).

Return Value *bDrawn*, a Boolean value, is nonzero if the string is drawn. A return value of zero means that one of the following errors occurred:

- The application-supplied output function returned zero.
- **TextOut** returned zero.
- Insufficient memory was available to create a memory bitmap for graying.

Notes The address passed as the *lpOutputFunc* parameter must be created using **MakeProcInstance**.

The function pointed to by *lpOutputFunc* must have the form:

lpOutputFunc (*hDC*, *lpData*, *nCount*) : *bSuccess*

where *hDC*, *lpData*, and *nCount* are the same as the parameters by the same name passed to **GrayString**.

The return value, a Boolean value, must be nonzero to indicate success, and zero if an error occurs.

DrawIcon (*hDC*, *X*, *Y*, *hIcon*) : *bDrawn*

Purpose This function draws an icon on the specified display, placing the icon's upper left corner at the location specified by *X* and *Y*.

Parameters *hDC* is a handle to a display context.

X and *Y* are short integer values specifying the logical coordinates of the upper left corner of the icon.

hIcon is a handle to an icon resource. The resource must have been previously loaded (using **LoadIcon**).

Return Value *bDrawn*, a Boolean value, is nonzero if the function is successful. Otherwise, it is 0.

SetPixel (*hDC*, *X*, *Y*, *rgbColor*) : *rgbActualColor*

Purpose This function sets the pixel at the point specified by *X* and *Y* to the closest approximation to the color specified by *rgbColor*. The point must be in the clipping region. If the point is not in the clipping region, the function is ignored.

Parameters *hDC* is a handle to a display context.

X and *Y* are short integer values specifying the logical coordinates of the point to be set.

rgbColor is an RGB color value specifying the color to paint the point.

Return Value *rgbActualColor* is an RGB color value specifying the color that the point is actually painted. This value can be different than *rgbColor* if an approximation of that color is used. If the function fails, e.g., the point is outside of clipping region, *rgbActualColor* is -1.

Note Not all devices support the **SetPixel** function. See the **GetDeviceCaps** function and the RC_BITBLT capability.

GetPixel (*hDC*, *X*, *Y*) : *rgbColor*

Purpose This function retrieves the RGB color value of the pixel at the point specified by *X* and *Y*. The point must be in the clipping region. If the point is not in the clipping region, the function is ignored.

Parameters *hDC* is a handle to a display context.

X and *Y* are short integer values specifying the logical coordinates of the point to be examined.

Return Value If the function is successful, *rgbColor*, an RGB color value, is the color of the given point. It is -1 if the coordinates do not specify a point in the clipping region.

Note Not all devices support the **GetPixel** function. See the **GetDeviceCaps** function and the RC_BITBLT capability.

FloodFill (*hDC, X, Y, rgbColor*) : *bFilled*

Purpose This function fills an area of the display surface with the current brush. The area is assumed to be bounded by the given *rgbColor*. The function begins at the point specified by *X* and *Y* and continues in all directions to the color boundary.

Parameters *hDC* is a handle to a display context.
X and *Y* are short integer values specifying the logical coordinates of the point to begin filling.
rgbColor is an RGB color value specifying the color of the boundary.

Return Value *bFilled*, a Boolean value, is nonzero if the function is successful. It is zero if the filling could not be completed, the given point has the boundary color *rgbColor*, or the point is outside the clipping region.

Note Not all devices support the **FloodFill** function. See the **GetDeviceCaps** function and the RC_BITBLT capability.

LineDDA (*X1, Y1, X2, Y2, lpLineFunc, lpData*)

Purpose This function computes all successive points in the line starting at the point specified by *X1* and *Y1* and ending at the point specified by *X2* and *Y2*. The end point is not included as part of the line. For each point on the line, **LineDDA** calls the application-supplied function pointed to by *lpLineFunc*, passing to the function the coordinates of the current point and the *lpData* parameter.

Parameters *X1* and *Y1* are short integer values specifying the logical coordinates of the first point.
X2 and *Y2* are short integer values specifying the logical coordinates of the last point.
lpLineFunc is a long pointer to an application-supplied function.
lpData is a long pointer to application-supplied data.

Return Value None.

Notes

The application-supplied function must have the form:

lpLineFunc(X, Y, lpData)

where *X* and *Y* are short integer coordinates of the current point, and *lpData* is a long pointer to the application-supplied data. The function can perform any task. It has no return value.

FillRgn (*hDC, hRgn, hBrush*) : *bFilled*

Purpose This function fills the region specified by *hRgn* with the brush specified by *hBrush*.

Parameters *hDC* is a handle to a display context.

hRgn is a handle to a region to be filled.

hBrush is a handle to the brush to be used to fill the region.

Return Value *bFilled*, a Boolean value, is nonzero if the function is successful. Otherwise, it is zero.

FrameRgn (*hDC, hRgn, hBrush, nWidth, nHeight*) : *bFramed*

Purpose This function draws a border around the region specified by *hRgn*, using the brush specified by *hBrush*. The *nWidth* parameter specifies the width of the border on vertical brush strokes; *nHeight* specifies the height on horizontal strokes.

Parameter *hDC* is a handle to a display context.

hRgn is a handle to a region to be enclosed in a border.

hBrush is a handle to the brush to be used to draw the border.

nWidth is a short integer number specifying the width of vertical brush strokes.

nHeight is a short integer number specifying the height of horizontal brush strokes.

Return Value *bFramed*, a Boolean value, is nonzero if the function is successful. Otherwise, it is zero.

InvertRgn (*hDC, hRgn*) : *bInverted*

Purpose This function inverts the colors in the region specified by *hRgn*.

Parameters *hDC* is a handle to a device context containing the region.
hRgn is a handle to a region to be filled.

Return Value *bInverted*, a Boolean value, is nonzero if the function is successful. Otherwise, it is zero.

PaintRgn (*hDC, hRgn*) : *bFilled*

Purpose This function fills the region specified by *hRgn* with the currently selected brush.

Parameters *hDC* is a handle to a device context containing the region.
hRgn is a handle to a region to be filled.

Return Value *bFilled*, a Boolean value, is nonzero if the function is successful. Otherwise, it is zero.

FillRect (*hDC, lpRect, hBrush*)

Purpose This function fills a given rectangle using the specified brush.

Parameters *hDC* is a handle to a display context.
lpRect is a long pointer to a **RECT** data structure.
hBrush is a handle to a brush used to fill the rectangle.

Return Value None.

FrameRect (*hDC, lpRect, hBrush*)

Purpose This function draws a border for the rectangle whose logical coordinates are pointed to by *lpRect*. The function uses the specified brush to draw the border.

Parameters *hDC* is a handle to a display context.
lpRect is a long pointer to a data structure having **RECT** type. The structure must contain the logical coordinates of the upper left and lower right corners of the rectangle.

hBrush is a handle to a brush to be used for framing the rectangle.

Return Value None.

InvertRect (*hDC, lpRect*)

Purpose This function inverts the display bits of the specified rectangle.

Parameters *hDC* is a handle to a display context.

lpRect is a long pointer to a data structure having **RECT** type.

Return Value None.

3.4 Drawing Object Functions

This section describes the functions used to create drawing objects. Drawing objects are used by GDI to create images on the display surface. They include pens, brushes, bitmaps, and fonts. Regions, also described in this section, are special objects that represent areas to draw instead of a style or color in which to draw an image.

GetStockObject (*nIndex*) : *hObject*

Purpose This function retrieves a handle to one of the several predefined stock pens, brushes, or fonts.

Parameters *nIndex* is a short integer value specifying the type of stock object desired. It can be any one of the following:

Value	Stock Object
WHITE_BRUSH	White Brush
LTGRAY_BRUSH	Light Gray Brush
GRAY_BRUSH	Gray Brush
DKGRAY_BRUSH	Dark Gray Brush
BLACK_BRUSH	Black Brush
HOLLOW_BRUSH	Hollow Brush

NULL_BRUSH	Null Brush
WHITE_PEN	White Pen
BLACK_PEN	Black Pen
NUL_PEN	Null Pen
ANSI_FIXED_FONT	ANSI Fixed System Font
OEM_FIXED_FONT	OEM-dependent Fixed Font
ANSI_VAR_FONT	ANSI Variable System Font
SYSTEM_FONT	System-dependent Fixed Font
DEVICE_DEFAULT_FONT	Device-dependent Font

Return Value If the function is successful, *hObject* is a handle to the logical object. Otherwise, it is NULL.

Notes The LTGRAY_BRUSH, GRAY_BRUSH, and DKGRAY_BRUSH objects should not be used as a background brush or for any other purpose in a window whose class does not specify CS_VREDRAW and CS_HREDRAW styles. Using a gray stock brush in such windows can lead to misalignment of brush patterns after a window is moved or sized. Stock brush origins cannot be adjusted (see the **SetBrushOrg** function).

CreatePen (*nPenStyle*, *nWidth*, *rgbColor*) : *hPen*

Purpose This function creates a logical pen having the specified style, width, and color. The pen can be subsequently selected as the current pen for any device.

Parameters *nPenStyle* is a short integer value specifying the pen style. It can be any one of the following:

Value	Pen Style
0	Solid
1	Dash
2	Dot
3	Dash and dot
4	Dash and two dots
5	Null

nWidth is a short integer value specifying the width of the pen in logical units.

rgbColor is an RGB color value specifying the color of the pen.

Return Value If the function is successful, *hPen* is a handle to a logical pen. Otherwise, it is NULL.

Note Pens with a physical width greater than 1 will always have either null or solid style.

CreatePenIndirect (*lpLogPen*) : *hPen*

Purpose This function creates a logical pen having the style, width, and color given in the data structure pointed to by *lpLogPen*.

Parameters *lpLogPen* is a long pointer to a data structure having **LOG-PEN** type.

Return Value If the function is successful, *hPen* is a handle to a logical pen object. Otherwise, it is NULL.

Note Pens with a physical width greater than 1 will always have either null or solid style.

CreateSolidBrush (*rgbColor*) : *hBrush*

Purpose This function creates a logical brush having the specified solid color. The brush can be subsequently selected as the current brush for any device.

Parameters *rgbColor* is an RGB color value specifying the color of the brush.

Return Value If the function is successful, *hBrush* is a handle to a logical brush. Otherwise, it is NULL.

CreateHatchBrush (*nIndex*, *rgbColor*) : *hBrush*

Purpose This function creates a logical brush having the specified hatched pattern and color. The brush can be subsequently selected as the current brush for any device.

Parameters *nIndex* is a short integer value specifying the hatch style of the brush. It can be any one of the following:

Value	Hatch Style
HS_HORIZONTAL	horizontal hatch
HS_VERTICAL	vertical hatch
HS_FDIAGONAL	45-degree upward hatch from left to right
HS_BDIAGONAL	45-degree downward hatch from left to right
HS_CROSS	horizontal and vertical cross-hatch
HS_DIAGCROSS	45-degree cross-hatch

rgbColor is an RGB color value specifying the foreground color of the brush; that is, the color of the hatches.

Return Value If the function is successful, *hBrush* is a handle to a logical brush. Otherwise, it is NULL.

CreatePatternBrush (*hBitmap*) : *hBrush*

Purpose This function creates a logical brush with the pattern specified by *hBitmap*. The brush can be subsequently selected as the current brush for any device that supports raster operations (see the **GetDeviceCaps** routine and the RC_BITBLT capability).

Parameters *hBitmap* is a handle to a bitmap. It is assumed to have been created using the **CreateBitmap**, **CreateBitmapIndirect**, or **CreateCompatibleBitmap**, function. The minimum size for a bitmap to be used in a fill pattern is 8 by 8.

Return Value If the function is successful, *hBrush* is a handle to a logical brush. Otherwise, it is NULL.

Note A pattern brush can be deleted with the **DeleteObject** function without affecting the associated bitmap. This means the bitmap can be used to create any number of pattern brushes.

CreateBrushIndirect (*lpLogBrush*) : *hBrush*

Purpose This function creates a logical brush having the style, color, and pattern given in the data structure pointed to by *lpLogBrush*. The brush can be subsequently selected as the current brush for any device.

Parameters *lpLogBrush* is a long pointer to a data structure having **LOGBRUSH** type.

Return Value If the function is successful, *hBrush* is a handle to a logical brush. Otherwise, it is NULL.

CreateBitmap (*nWidth*, *nHeight*, *nPlanes*, *nBitCount*, *lpBits*) : *hBitmap*

Purpose This function creates a bitmap having the specified width, height, and bit pattern. The bitmap can subsequently be selected as the current bitmap for a memory display using the **SelectObject** function. The memory display must have a compatible planes/pixels format.

Although a bitmap cannot be copied directly to a display device, the **BitBlt** function can copy it from a memory display context in which it is the current bitmap to any compatible device.

Parameters *nWidth* is a short integer value specifying the width in pixels of the bitmap.

nHeight is a short integer value specifying the height in pixels of the bitmap.

nPlanes is a short integer value specifying the number of color planes in the bitmap. Each plane has *nWidth*nHeight*nBitCount* bits.

nBitCount is a short integer value specifying the number of color bits per display pixel.

lpBits is a long pointer to a short integer array containing the initial bitmap bit values, or it is the long integer value 0 specifying an uninitialized bitmap.

Return Value If the function is successful, *hBitmap* is a handle to a bitmap. Otherwise, it is NULL.

CreateBitmapIndirect (*lpBitmap*) : *hBitmap*

Purpose This function creates a bitmap having the width, height, and bit pattern given in the data structure pointed to by *lpBitmap*.

Although a bitmap cannot be directly selected for a display device, it can be selected as the current bitmap for a memory display and copied to any compatible display device by using the **BitBlt** function.

Parameters *lpBitmap* is a long pointer to a data structure having **BITMAP** type.

Return Value If the function is successful, *hBitmap* is a handle to a bitmap. Otherwise, it is NULL.

CreateCompatibleBitmap (*hDC*, *nWidth*, *nHeight*) : *hBitmap*

Purpose This function creates a bitmap that is compatible with the device specified by *hDC*. The bitmap has the same number of color planes or bits per pixel format as the specified device. It can be selected as the current bitmap for any memory display that is compatible with *hDC*.

If *hDC* is a memory display context, the bitmap returned has the same format as the currently selected bitmap in that display context.

When a memory display context is created, GDI automatically selects a monochrome stock bitmap for it.

Since a color memory display context can have either color or monochrome bitmaps selected, the format of the bitmap returned by **CreateCompatibleBitmap** is not always the same; however, the format of a compatible bitmap for a non-memory display context is always in the format of the device.

Parameters *hDC* is a handle to a display context.

nWidth is a short integer value specifying the width in bits of the bitmap.

nHeight is a short integer value specifying the height in bits of the bitmap.

Return Value If the function is successful, *hBitmap* is a handle to a bitmap. Otherwise, it is NULL.

SetBitmapBits (*hBitmap*, *dwCount*, *lpBits*) : *dwCopied*

Purpose This function sets the bits of a bitmap to the bit values given by *lpBits*.

Parameters *hBitmap* is a handle to the bitmap to be set.

dwCount is an unsigned long integer value specifying the number of bytes pointed to by *lpBits*.

lpBits is a long pointer to a short integer array of bits.

Return Value *dwCopied*, an unsigned long integer value, specifies the number of bytes actually copied.

GetBitmapBits (*hBitmap*, *dwCount*, *lpBits*) : *dwCopied*

Purpose This function copies the bits of the specified bitmap into the buffer pointed to by *lpBits*. The *dwCount* parameter specifies the number of bytes to copy to the buffer. The **GetObject** function should be used to determine the correct *dwCount* value for the given bitmap.

Parameters *hBitmap* is a handle to a bitmap.

dwCount is an unsigned long integer value specifying the number of bytes to be copied.

lpBits is a long pointer to the buffer to receive the bitmap. The bitmap is an array of short integers.

Return Value *dwCopied*, an unsigned long integer value, is the actual number of bytes in the bitmap. It is 0 if there is an error.

SetBitmapDimension (*hBitmap*, *X*, *Y*) : *ptOldDimensions*

Purpose This routine associates a width and height to a bitmap in 0.1 millimeter units. These values are not used internally by GDI, **GetBitmapDimension** can be used to retrieve them.

Parameters *hBitmap* is a handle to a bitmap.

X is a short integer specifying the width of the bitmap in 0.1 millimeter units.

Y is a short integer specifying the height of the bitmap in 0.1 millimeter units.

Return Value *ptOldDimensions* is a long integer value specifying the previous bitmap dimensions. The height is in the high-order word, width is in the low-order word.

GetBitmapDimension (*hBitmap*) : *ptDimensions*

Purpose This routine returns the width and height of the bitmap specified by *hBitmap*. The height and width is assumed to have been previously set using the **SetBitmapDimension** function.

Parameters *hBitmap* is a handle to a bitmap.

Return Value *ptDimensions* is a long integer value specifying the width and height of the bitmap in 0.1 millimeter units. The height is in the high-order word, width is in the low-order word. If the bitmap width and height have not been set using **SetBitmapDimension**, *ptDimensions* is zero.

**CreateFont (*nHeight*, *nWidth*, *nEscapement*, *nOrientation*, *nWeight*,
cItalic, *cUnderline*, *cStrikeOut*, *nCharSet*, *nOutputPrecision*,
nClipPrecision, *cQuality*, *cPitchAndFamily*, *lpFacename*)
: *hFont***

Purpose This function creates a logical font having the specified characteristics. The logical font can subsequently be selected as the current font for any device.

Parameters *nHeight* is a short integer value specifying the desired height (in logical units) of the font. The height of a font can be specified in three ways. If *nHeight* is greater than zero, it is transformed into device units and matched against the cell height of the available fonts. If *nHeight* is zero, a reasonable default size is used. If *nHeight* is less than zero, it is transformed into device units and the absolute value is matched against the character height of the available fonts. For all height comparisons, the font mapper looks for the largest font which does not exceed the requested size, and if there is no such font, looks for the smallest font available.

nWidth is a short integer value specifying the average width of characters in the font in logical units. If *nWidth* is zero, the aspect ratio of the device will be matched against the digitization aspect ratio of the available fonts looking for the closest match by absolute value of the difference.

nEscapement is a short integer value specifying the angle (in tenths of degrees) of each line of text written in the font (relative to the bottom of the page).

nOrientation is a short integer value specifying the angle (in tenths of degrees) of each character's baseline relative to the bottom of the page.

nWeight is a short integer value specifying the desired weight of the font in the range 0 to 1000 (for example, 400 is normal, 700 is bold). If 0, a default weight is used.

cItalic is an 8-bit flag specifying whether or not the font is italic.

cUnderline is an 8-bit flag specifying whether or not the font is underlined.

cStrikeOut is an 8-bit flag specifying whether or not characters in the font are struck out.

nCharSet is a short integer value specifying the desired character set. It can be one of the following:

ANSI_CHARSET	Windows standard ANSI character set
OEM_CHARSET	System-specific character set

nOutputPrecision is a short integer value specifying the desired output precision. It can be any one of the following:

OUT_DEFAULT_PRECIS
OUT_STRING_PRECIS
OUT_CHARACTER_PRECIS
OUT_STROKE_PRECIS

nClipPrecision is a short integer value specifying the desired clipping precision. It can be any one of the following:

CLIP_DEFAULT_PRECIS
CLIP_CHARACTER_PRECIS
CLIP_STROKE_PRECIS

cQuality is an 8-bit flag specifying the desired output quality. It can be any one of the following:

Value	Meaning
PROOF_QUALITY	The character quality of the font is more important than exact matching of the logical font attributes. For GDI fonts, scaling is disabled and the font closest in size is chosen. Although the chosen font size may not be mapped exactly when PROOF_QUALITY is used, the quality of the font is high and there is no degradation of appearance. Bold, italic, underline, and strikeout are synthesized if needed.
DRAFT_QUALITY	The appearance of the font is less important than if PROOF_QUALITY is used. For GDI fonts, scaling is enabled, with the result that more font sizes are available, but the quality may be lower. Bold, italic, underline, and strikeout are synthesized if needed.
DEFAULT_QUALITY	The appearance of the font does not matter.

cPitchAndFamily is an 8-bit flag specifying the font pitch and family. The low 2 bits specify the pitch of the font and can be any one of the following:

DEFAULT_PITCH
FIXED_PITCH
VARIABLE_PITCH

The four high-order bits of the field specify the font family and can be any one of the following:

FF_DONTCARE
 FF_ROMAN
 FF_SWISS
 FF_MODERN
 FF_SCRIPT
 FF_DECORATIVE

lpFacename is a long pointer to a null-terminated ASCII string specifying the facename of the font. The **EnumFonts** function can be used to enumerate the facenames of all currently available fonts.

Return Value If the function is successful, *hFont* is a handle to a logical font. Otherwise, it is NULL.

Notes **CreateFont** creates a logical font having all the specified characteristics. When the font is selected using **SelectObject**, GDI's font mapper attempts to match the logical font with an existing physical font. If it fails to find an exact match, it provides an alternate whose characteristics match as many of the requested characteristics as possible. See the *Windows Programming Guide* for a description of the font mapper.

CreateFontIndirect (*lpLogFont*) : *hFont*

Purpose This function creates a logical font having the characteristics given in the data structure pointed to by *lpLogFont*. The font can be subsequently selected as the current font for any device.

Parameters *lpLogFont* is a long pointer to a data structure having **LOGFONT** type.

Return Value If the function is successful, *hFont* is a handle to a logical font. Otherwise, it is NULL.

Notes **CreateFontIndirect** creates a logical font having all the specified characteristics. When the font is selected using **SelectObject**, GDI's font mapper attempts to match the logical font with an existing physical font. If it fails to find an exact font, it provides an alternate whose characteristics match as many of the requested characteristics as possible. See the *Windows Programming Guide* for a description of the font mapper.

DeleteObject (*hObject*) : *bDeleted*

Purpose This function deletes a logical pen, brush, font, bitmap, or region from memory by freeing all system storage associated with the object. After deletion, the handle *hObject* is no longer valid.

Parameters *hObject* is a handle to a logical pen, brush, font, bitmap, or region.

Return Value *bDeleted*, a Boolean value, is nonzero if the specified object is deleted. It is zero if *hObject* is not a valid handle or is currently selected into a display context.

Notes The object to be deleted should not be currently selected in a display context, unless the display context is deleted before the object is deleted.
When a pattern brush is deleted, the bitmap associated with the brush is not deleted. The bitmap must be deleted on its own.

3.5 Selection Functions

This section describes the functions used to select the current pen, brush, bitmap, or font for a given display context. The selected object is the default drawing object used by GDI whenever an output function does not specify an explicit drawing object.

SelectObject (*hDC*, *hObject*) : *hOldObject*

Purpose This function selects the logical object specified by *hObject* as the currently selected object of the specified display context. The new object replaces the previous object of the same type. For example, if *hObject* is the handle to a logical pen, **SelectObject** replaces the currently selected pen with the pen specified by *hObject*.
Selected objects are the default objects used by the GDI output functions to draw lines, fill interiors, write text, and clip output to specific areas of the display surface. Although a display context can have five selected objects (i.e., pen, brush, font, bitmap, and region), no more than one object of any given type can be selected at any given time.

Parameters *hDC* is a handle to a display context.
hObject is a handle to a logical object that has been created using one of the following functions:

Object	Function
Pen	Must be created using CreatePen or CreatePenIndirect .
Brush	Must be created using CreateSolidBrush , CreateHatchBrush , CreatePatternBrush , or CreateBrushIndirect .
Font	Must be created using CreateFont or CreateFontIndirect .
Bitmap	Must be created using CreateBitmap , CreateBitmapIndirect , or CreateCompatibleBitmap . Bitmaps can be selected for memory display contexts only, and for only one display context at a time.
Region	Must be created using CreateRectRgn , CreateEllipticRgn , CreatePolygonRgn , CombineRgn , CreateRectRgnIndirect , or CreateEllipticRgnIndirect .

Return Value *hOldObject* is a handle to the object being replaced by *hObject*. It is NULL if there is an error.

If *hDC* specifies a metafile, *hOldObject* is nonzero if the function is successful. Otherwise, it is zero.

SelectClipRgn (*hDC*, *hRgn*) : *nRgnType*

Purpose This function selects the given region as the current clipping region for the specified display context. Only a copy of the selected region is used. The region itself can be selected for any number of other display contexts or be deleted.

Parameters *hDC* is a handle to a display context.
hRgn is a handle to a region.

Return Value *nRgnType* is a short integer value specifying the region's type. It can be any one of the following:

ERROR
NULLREGION
SIMPLEREGION
COMPLEXREGION

GetObject (*hObject*, *nCount*, *lpObject*) : *nCopied*

Purpose This function fills a buffer with the logical data that defines the logical object specified by *hObject*. **GetObject** copies *nCount* bytes of data to the buffer pointed to by *lpObject*. The function returns data structures of the type **LOGPEN**, **LOGBRUSH**, **LOGFONT**, or **BITMAP** depending on the logical object. The buffer must be sufficiently large to receive the data.

Parameters *hObject* is a handle to a logical pen, brush, font, or bitmap. *nCount* is a short integer value specifying the number of bytes to copy to the buffer.

lpObject is a long pointer to the buffer to receive the information. It must be a data structure of type **LOGPEN**, **LOGBRUSH**, **LOGFONT**, or **BITMAP**, depending on the logical object given.

Return Value *nCopied*, a short integer value, is the actual number of bytes retrieved. It is 0 if there is an error.

Notes If *hObject* specifies a bitmap, the function returns only the width, height, and color format information of the bitmap. The actual bits must be retrieved using the **GetBitmap-Bits** function.

3.6 Display Context Attribute Functions

This section describes the functions used to set or examine the current attributes of a display context. These attributes include the context's current background color, text color, and mapping mode.

The default values for the display context attributes are:

Attribute	Default Selection
Brush	WHITE_BRUSH
Pen	BLACK_PEN
Bitmap	No default
Font	SYSTEM_FONT
Current Pen Position	(0,0)
Brush Origin	(0,0)
Relabs Flag	ABSOLUTE
Text Color	Black
Background Color	White
Background Mode	OPAQUE
Drawing Mode	R2_COPYPEN
Stretching Mode	BLACKONWHITE
Mapping Mode	MM_TEXT
Intercharacter Spacing	0
Polygon Filling Mode	ALTERNATE
Color Table	Entries are set to represent an even distribution of the full range of available colors. The exact default values depend on the device. For example, raster devices have black as the first entry, white as the last; printers have white as the first entry, black as the last.
Clipping Region	The whole display surface
Window Origin	(0, 0)
Window Extents	(1, 1)
Viewport Origin	(0, 0)
Viewport Extents	(1 , 1)

SetRelAbs (*hDC*, *nRelAbsMode*) : *nOldRelAbsMode*

Purpose This function sets the **relabs** flag. This flag specifies whether the coordinates in GDI functions are relative to the origin of the given device (absolute), or relative to the current position (relative).

Parameters *hDC* is a handle to a display context.

nRelAbsMode is a short integer flag specifying the current mode. It can be any one of the following:

ABSOLUTE
RELATIVE

Relabs flag set to absolute
Relabs flag set to relative

Return Value *nOldRelAbsMode* is a short integer value specifying the previous setting of the **relabs** flag. It can be ABSOLUTE or RELATIVE. It is NULL if *hDC* is not a valid display context.

Notes The **relabs** flag affects the **LineTo** and **PolyLine** functions.

GetRelAbs (*hDC*) : *nRelAbsMode*

Purpose This function retrieves the **relabs** flag. This flag specifies whether the coordinates in GDI functions are relative to the origin of the given device (absolute), or relative to the current position (relative).

Parameters *hDC* is a handle to a display context.

Return Value *nRelAbsMode* is a short integer flag specifying the current mode. It can be ABSOLUTE or RELATIVE. It is NULL if *hDC* is not a valid display context.

SetBkColor (*hDC*, *rgbColor*) : *rgbOldColor*

Purpose This function sets the current background color to the color specified by *rgbColor*, or to the nearest logical color if the device cannot represent *rgbColor*.

If the background mode is OPAQUE, GDI uses the background color to fill the gaps between styled lines, gaps between hatched lines in brushes, and character cells. GDI also uses the background color when converting bitmaps from color to monochrome and vice versa.

The background mode is set by the **SetBkMode** function. See the **BitBlt** and **StretchBlt** functions for color bitmap conversions.

Parameters *hDC* is a handle to a display context.

rgbColor is an RGB color value specifying the new background color.

Return Value *rgbOldColor*, is an RGB color value specifying the previous background color. If an error occurs, *rgbOldColor* is 80000000H.

GetBkColor (*hDC*) : *rgbColor*

Purpose This function returns the current background color of the specified device.

Parameters *hDC* is a handle to a display context.

Return Value *rgbColor*, an RGB color value, is the current background color.

SetBkMode (*hDC*, *nBkMode*) : *nOldBkMode*

Purpose This function sets the background mode used with text, hatched brushes, and line styles. The background mode defines whether or not GDI should overwrite existing background colors on the display surface before drawing text, hatched brushes, or any pen style that is not a solid line.

Parameters *hDC* is a handle to a display context.

nBkMode is a short integer value specifying the background mode. It can be one of the following:

Mode	Meaning
TRANSPARENT	Background remains untouched.
OPAQUE	Background is filled with the current background color before the text, hatched brush, or pen is drawn.

Return Value *nOldBkMode* is a short integer value specifying the previous background mode. It can be TRANSPARENT or OPAQUE.

GetBkMode (*hDC*) : *nBkMode*

Purpose This function returns the background mode of the specified device. The background mode is used with text, hatched brushes, and any pen style that is not a solid line.

Parameters *hDC* is a handle to a display context.

Return Value *nBkMode* is a short integer value specifying the current background mode. It can be TRANSPARENT or OPAQUE.

SetTextColor (*hDC*, *rgbColor*) : *rgbOldColor*

Purpose This function sets the text color to the color specified by *rgbColor*, or to the nearest logical color if the device cannot represent *rgbColor*. GDI uses the text color to draw the face of each character written by the **TextOut** function. GDI also uses the text color when converting bitmaps from color to monochrome and vice versa.

The background color for a character is specified by the **SetBkColor** and **SetBkMode** functions. See the **BitBlt** and **StretchBlt** functions for color bitmap conversions.

Parameters *hDC* is a handle to a display context.

rgbColor is an RGB color value specifying the color of the text.

Return Value *rgbOldColor* is an RGB color value specifying the previous text color.

GetTextColor (*hDC*) : *rgbColor*

Purpose This function retrieves the current text color. The text color defines the foreground color of characters drawn using the **TextOut** function.

Parameters *hDC* is a handle to a display context.

Return Value *rgbColor*, an RGB color value, is the current text color.

SetROP2 (*hDC*, *nDrawMode*) : *nOldDrawMode*

Purpose This function sets the current drawing mode. GDI uses the drawing mode to combine pens and interiors of filled objects with the colors already on the display surface. The mode specifies how the color of the pen or interior and the color already on the display surface are combined to yield a new color.

The drawing mode is for raster devices only; it is not available on vector devices. See the **GetDeviceCaps** function and the RC_BITBLT capability.

- Parameters* *hDC* is a handle to a display context.
- nDrawMode* is a short integer value specifying the new drawing mode. It can be any one of the values given in Table 3.1.
- Return Value* *nOldDrawMode* is a short integer value specifying the previous drawing mode. It can be any one of the *nDrawMode* values given in Table 3.1.

GetROP2 (*hDC*) : *nDrawMode*

- Purpose* This function retrieves the current drawing mode. The drawing mode specifies how the pen or interior color and the color already on the display surface are combined to yield a new color.
- Parameters* *hDC* is a handle to a display context for a raster device.
- Return Value* *nDrawMode* is a short integer value that specifies the drawing mode. It can be any one of the values given in Table 3.1.

Table 3.1
Raster Operation Values

Value	Meaning
R2_BLACK	Pixel is always black.
R2_NOTMERGEPE	Pixel is the inverse of the R2_MERGEPE color.
R2_MASKNOTPE	Pixel is a combination of the colors common to both the display and the inverse of the pen.
R2_NOTCOPYPE	Pixel is the inverse of the pen color.
R2_MASKPENNOT	Pixel is a combination of the colors common to both the pen and the inverse of the display.
R2_NOT	Pixel is the inverse of the display color.

Table 3.1 (continued)

Value	Meaning
R2_XORPEN	Pixel is a combination of the colors in the pen and in the display, but not in both.
R2_NOTMASKPEN	Pixel is the inverse of R2_MASKPEN color.
R2_MASKPEN	Pixel is a combination of the colors common to both the pen and the display.
R2_NOTXORPEN	Pixel is the inverse of R2_XORPEN color.
R2_NOP	Pixel remains unchanged.
R2_MERGENOTPEN	Pixel is a combination of the display and the inverse of the pen color.
R2_COPYPEN	Pixel is the pen color.
R2_MERGEPPENNOT	Pixel is a combination of the pen and the inverse of the display color.
R2_MERGEPPEN	Pixel is a combination of the pen and the display color.
R2_WHITE	Pixel is always white.

SetStretchBltMode (*hDC*, *nStretchMode*) : *nOldStretchMode*

Purpose This function sets the stretching mode for the **StretchBlt** function. The stretching mode defines which scan lines and/or columns **StretchBlt** eliminates when contracting a bitmap.

The *nStretchMode* can be:

Mode	Meaning
WHITEONBLACK	OR in the “eliminated” lines. This mode preserves white pixels at the expense of black pixels by ORing the lines to be eliminated with the remaining lines.
BLACKONWHITE	AND in the “eliminated” lines. This mode preserves black pixels at the expense of white pixels by ANDing the eliminated lines with those remaining.

COLORONCOLOR

Throw out the “eliminated” lines. This mode throws all eliminated lines out without trying to preserve their information.

The WHITEONBLACK and BLACKONWHITE modes are typically used to preserve foreground pixels in monochrome bitmaps. The COLORONCOLOR mode is typically used to preserve color in color bitmaps.

Parameters

hDC is a handle to a display context.

nStretchMode is a short integer value specifying the stretching mode.

Return Value

nOldStretchMode, is a short integer value specifying the previous stretching mode. It can be WHITEONBLACK, BLACKONWHITE, or COLORONCOLOR.

GetStretchBltMode (*hDC*) : *nStretchMode**Purpose*

This function retrieves the current stretching mode. The stretching mode defines how information is to be added or removed from bitmaps that are stretched or compressed using the **StretchBlt** function.

Parameters

hDC is a handle to a display context.

Return Value

nStretchMode is a short integer number specifying the current stretching mode. It can be WHITEONBLACK, BLACKONWHITE, or COLORONCOLOR. See the **SetStretchBltMode** function for an explanation of each mode.

SetPolyFillMode (*hDC*, *nPolyFillMode*) : *nOldPolyFillMode**Purpose*

This function sets the polygon filling mode for the GDI functions that use the polygon algorithm to compute interior points. The *nPolyFillMode* can be:

Mode	Meaning
ALTERNATE	Selects alternate mode
WINDING	Selects winding number mode

In general, the modes differ only in cases where a complex, overlapping polygon must be filled (e.g., a five-sided polygon that forms a five-pointed star with a pentagon in the

center). In such cases, ALTERNATE mode fills every other enclosed region within the polygon (i.e., the points of the star), but WINDING mode fills all regions (i.e., the points and the pentagon). To fill all regions, the WINDING mode causes GDI to compute and draw a border that encloses the polygon but does not overlap. For example, in the WINDING mode, the five-sided polygon that forms the star is drawn as a ten-sided polygon with no overlapping sides; the resulting star is filled.

- Parameters* *hDC* is a handle to a display context.
nPolyFillMode is a short integer value specifying the filling mode.
- Return Value* *nOldPolyFillMode* is a short integer value specifying the previous filling mode. It is 0 if there is an error.

GetPolyFillMode (*hDC*) : *nPolyFillMode*

- Purpose* This function retrieves the current polygon filling mode.
Parameters *hDC* is a handle to a display context.
Return Value *nPolyFillMode* is a short integer value specifying the polygon filling mode. It can be one of the following:

Mode	Meaning
ALTERNATE	Alternate mode
WINDING	Winding number mode

See the **SetPolyFillMode** function for a description of these modes.

SetMapMode (*hDC*, *nMapMode*) : *nOldMapMode*

- Purpose* This function sets the mapping mode of the specified display context. The mapping mode defines the unit of measure used in transforming logical units into device units, and also defines the orientation of the device's x- and y-axes. GDI uses the mapping mode to convert logical coordinates into the appropriate device coordinates.
- Parameters* *hDC* is a handle to a display context.

nMapMode is a short integer value specifying the mapping mode. It can be any one of the following:

Mode	Meaning
MM_TEXT	Each logical unit is mapped to one device pixel. Positive x is to the right; positive y is down.
MM_LOMETRIC	Each logical unit is mapped to one tenth of a millimeter (0.1 mm). Positive x is to the right; positive y is up.
MM HIMETRIC	Each logical unit is mapped to one one-hundredth of a millimeter (0.01 mm). Positive x is to the right; positive y is up.
MM LOENGLISH	Each logical unit is mapped to one one-hundredth of an inch (0.01 inches). Positive x is to the right; positive y is up.
MM_HIENGLISH	Each logical unit is mapped to one one-thousandth of an inch (0.001 inches). Positive x is to the right; positive y is up.
MM_TWIPS	Each logical unit is mapped to one twentieth of a printer's point (1/1440 inches). Positive x is to the right; positive y is up.

MM_ISOTROPIC

Logical units are mapped to arbitrary units with equally scaled axes; that is, one unit along the x-axis is equal to one unit along the y-axis. The **SetWindowExt** and **SetViewportExt** functions must be used to specify the desired units and the orientation of the axes. GDI makes adjustments as necessary to ensure that the x and y units remain the same size.

MM_ANISOTROPIC

Logical units are mapped to arbitrary units with arbitrarily scaled axes. The **SetWindowExt** and **SetViewportExt** functions must be used to specify the desired units, orientation, and scaling.

Return Value *nOldMapMode* is a short integer value specifying the previous mapping mode.

Notes **MM_TEXT** mode allows applications to work in device pixels, whose size will vary from device to device.

The **MM_LOMETRIC**, **MM_HIMETRIC**, **MM_LOENGLISH**, **MM_HIENGLISH**, and **MM_TWIPS** modes are useful for applications that need to draw in physically meaningful units (such as inches or millimeters).

MM_ISOTROPIC mode ensures a 1:1 aspect ratio, which is useful when preserving the exact shape of an image is important.

MM_ANISOTROPIC mode allows the x and y coordinates to be adjusted independently.

GetMapMode (*hDC*) : *nMapMode*

Purpose This function retrieves the current mapping mode. See **SetMapMode** for a description of the mapping modes.

Parameters *hDC* is a handle to a display context.

Return Value *nMapMode* is a short integer value specifying the mapping mode.

SetWindowOrg (*hDC*, *X*, *Y*) : *ptOldOrigin*

Purpose This function sets the window origin of the specified display context. The window, along with the display context viewport, defines how GDI maps points in the logical coordinate system to points in the device coordinate system.

The window origin marks the point in the logical coordinate system from which GDI maps the viewport origin, a point in the device coordinate system specified by the **SetWindowOrg** function. GDI maps all other points by following the same process required to map the window origin into the viewport origin. For example, all points in a circle around the point at the window origin will be in a circle around the point at the viewport origin. Similarly, all points in a line that passes through the window origin will be in a line that passes through the viewport origin.

Parameters *hDC* is a handle to a display context.

X and *Y* are short integer values specifying the x and y coordinates (in logical units) of the origin of the window.

Return Value *ptOldOrigin* is a long integer value specifying the previous origin of the window in logical coordinates. The y coordinate is in the high-order word; the x coordinate is in the low-order word.

GetWindowOrg (*hDC*) : *ptOrigin*

Purpose This function retrieves the x and y coordinates of the origin of the window associated with the specified display context.

Parameters *hDC* is a handle to a display context.

Return Value *ptOrigin* is a long integer value containing the origin in logical coordinates. The y coordinate is in the high-order word; the x coordinate in the low-order word.

SetWindowExt (*hDC*, *X*, *Y*) : *ptOldExtents*

Purpose

This function sets the x and y extents of the window associated with the specified display context. The window, along with the display context viewport, defines how GDI maps points in the logical coordinate system to points in the device coordinate system.

The x and y extents of the window define how much GDI must stretch or compress units in the logical coordinate system to fit units in the device coordinate system. For example, if the x extent of the window is 2 and the x extent of the viewport is 4, GDI maps 2 logical units (measured from the x axis) into 4 device units. Similarly, if the y extent of the window is 2 and the y extent of the viewport is -1, GDI maps 2 logical units (measured from the y axis) into 1 device unit.

The extents also define the relative orientation of the x and y axes in both coordinate systems. If the signs of matching window and viewport extents are the same, the axes have the same orientation. If signs are different, the orientation is reversed. For example, if the y extent of the window is 2 and the y extent of the viewport is -1, GDI maps the positive y axis in the logical coordinate system to the negative y axis in the device coordinate system. If the x extents are 2 and 4, GDI maps the positive x axis in the logical coordinate system to the positive x axis in the device coordinate system.

Parameters

hDC is a handle to a display context.

X and *Y* are short integer values specifying the extents of the window in logical units.

Return Value

ptOldExtents is a long integer value specifying the previous extents of the window in logical units. The previous y extent is in the high-order word; the previous x extent in the low-order word. On an error, *ptOldExtents* is zero.

Notes

When the following mapping modes are set, calls to **SetWindowExt** and **SetViewportExt** are ignored:

MM_TEXT
MM_LOMETRIC
MM_HIMETRIC
MM_LOENGLISH
MM_HIENGLISH
MM_TWIPS

When **MM_ISOTROPIC** mode is set, GDI makes adjustments to the window or viewport extents as necessary after calls to **SetWindowExt** or **SetViewportExt** to ensure that the x and y units remain the same size.

GetWindowExt (*hDC*) : *ptExtents*

Purpose This function retrieves the x and y extents of the window associated with the specified display context.

Parameters *hDC* is a handle to a display context.

Return Value *ptExtents* is a long integer value containing the x and y extents in logical units. The y extent is in the high-order word; the x extent in the low-order word.

SetViewportOrg (*hDC*, *X*, *Y*) : *ptOldOrigin*

Purpose This function sets the viewport origin of the specified display context. The viewport, along with the display context window, defines how GDI maps points in the logical coordinate system to points in the coordinate system of the actual device. In other words, they define how GDI converts logical coordinates to device coordinates.

The viewport origin marks the point in the device coordinate system to which GDI maps the window origin, a point in the logical coordinate system specified by the **SetWindowOrg** function. GDI maps all other points by following the same process required to map the window origin into the viewport origin. For example, all points in a circle around the point at the window origin will be in a circle around the point at the viewport origin. Similarly, all points in a line that passes through the window origin will be in a line that passes through the viewport origin.

Parameters *hDC* is a handle to a display context.

X and *Y* are short integer values specifying the x and y coordinates (in device units) of the origin of the viewport. The values must be within the range of the device coordinates system.

Return Value *ptOldOrigin* is a long integer value specifying the previous origin of the viewport in device coordinates. The y coordinate is in the high-order word; the x coordinate is in the low-order word.

GetViewportOrg (*hDC*) : *ptOrigin*

Purpose This function retrieves the x and y coordinates of the origin of the viewport associated with the specified display context.

Parameters *hDC* is a handle to a display context.

Return Value *ptOrigin* is a long integer value containing the origin in device coordinates. The y coordinate is in the high-order word; the x coordinate is in the low-order word.

ScaleWindowExt (*hDC*, *Xnum*, *Xdenom*, *Ynum*, *Ydenom*) : *ptOldExtents*

Purpose This function modifies the window extents relative to the current values. The formulas are:

$$\begin{aligned}x_{\text{NewWE}} &= (\text{xOldWE} * \text{Xnum}) / \text{Xdenom} \\y_{\text{NewWE}} &= (\text{yOldWE} * \text{Ynum}) / \text{Ydenom}\end{aligned}$$

The new extent is calculated by multiplying the current extents by the given numerator and then dividing by the given denominator.

Parameters *hDC* is a handle to a display context.

Xnum is a short integer specifying the amount to multiply the current x extent.

Xdenom is a short integer specifying the amount to divide the current x extent.

Ynum is a short integer specifying the amount to multiply the current y extent.

Ydenom is a short integer specifying the amount to divide the current y extent.

Return Value *ptOldExtents* is a long integer value specifying the previous window extents in logical units. The previous y extent is in the high-order word; the x extent is in the low-order word.

ScaleViewportExt (*hDC*, *Xnum*, *Xdenom*, *Ynum*, *Ydenom*)
: *ptOldExtents*

Purpose This function modifies the viewport extents relative to the current values. The formulas are:

$$\begin{aligned}x_{\text{NewVE}} &= (x_{\text{OldVE}} * \text{Xnum}) / \text{Xdenom} \\y_{\text{NewVE}} &= (y_{\text{OldVE}} * \text{Ynum}) / \text{Ydenom}\end{aligned}$$

The new extent is calculated by multiplying the current extents by the given numerator and dividing by the given denominator.

Parameters *hDC* is a handle to a display context.

Xnum is a short integer specifying the amount to multiply the current x extent.

Xdenom is a short integer specifying the amount to divide the current x extent.

Ynum is a short integer specifying the amount to multiply the current y extent.

Ydenom is a short integer specifying the amount to divide the current y extent.

Return Value *ptOldExtents* is a long integer value specifying the previous viewport extents in device units. The previous y extent is in the high-order word; the x extent is in the low-order word.

OffsetWindowOrg (*hDC*, *X*, *Y*) : *ptOldOrgs*

Purpose This function modifies the viewport origin relative to the current values. The formulas are:

$$\begin{aligned}x_{\text{NewWO}} &= x_{\text{OldWO}} + X \\y_{\text{NewWO}} &= y_{\text{OldWO}} + Y\end{aligned}$$

The new origin is the sum of the current origin and the *X* and *Y* values.

Parameters *hDC* is a handle to a display context.

X is a short integer specifying the amount (in logical units) to add to the current origin's x coordinate.

Y is a short integer specifying the amount (in logical units) to add to the current origin's *y* coordinate.

Return Value *ptOldOrgs* is a long integer value specifying the previous window origin in logical coordinates. The previous *y* coordinate is in the high-order word; the *x* coordinate is in the low-order word.

OffsetViewportOrg (*hDC*, *X*, *Y*) : *ptOldOrgs*

Purpose This function modifies the viewport origin relative to the current values. The formulas are:

$$\begin{aligned}x_{\text{NewVO}} &= x_{\text{OldVO}} + X \\y_{\text{NewVO}} &= y_{\text{OldVO}} + Y\end{aligned}$$

The new origin is the sum of the current origin and the *X* and *Y* values.

Parameters *hDC* is a handle to a display context.

X is a short integer specifying the amount (in device units) to add to the current origin's *x* coordinate.

Y is a short integer specifying the amount (in device units) to add to the current origin's *y* coordinate.

Return Value *ptOldOrgs* is a long integer value specifying the previous viewport origin in device coordinates. The previous *y* coordinate is in the high-order word; the *x* coordinate is in the low-order word.

SetViewportExt (*hDC*, *X*, *Y*) : *ptOldExtents*

Purpose This function sets the *x* and *y* extents of the viewport of the specified display context. The viewport, along with the display context window, defines how GDI maps points in the logical coordinate system to points in the coordinate system of the actual device. In other words, the viewport and display context window define how GDI converts logical coordinates to device coordinates.

The *x* and *y* extents of the viewport define how much GDI must stretch or compress units in the logical coordinate system to fit units in the device coordinate system. For example, if the *x* extent of the window is 2 and the *x* extent of the viewport is 4, GDI maps two logical units (measured from

the x axis) into four device units. Similarly, if the y extent of the window is 2 and the y extent of the viewport is -1, GDI maps 2 logical units (measured from the y axis) into one device unit.

The extents also define the relative orientation of the x and y axes in both coordinate systems. If the signs of matching window and viewport extents are the same, the axes have the same orientation. If signs are different, the orientation is reversed. For example, if the y extent of the window is 2 and the y extent of the viewport is -1, GDI maps the positive y axis in the logical coordinate system to the negative y axis in the device coordinate system. If the x extents are 2 and 4, GDI maps the positive x axis in the logical coordinate system to the positive x axis in the device coordinate system.

Parameters *hDC* is a handle to a display context.

X and *Y* are short integer values specifying the extents of the viewport in device units.

Return Value *ptOldExtents* is a long integer value specifying the previous extents of the viewport in device units. The previous y extent is in the high-order word; the previous x extent is in the low-order word. On an error, *ptOldExtents* is zero.

Notes When the following mapping modes are set, calls to **SetWindowExt** and **SetViewportExt** are ignored:

MM_TEXT
MM_LOMETRIC
MM_HIMETRIC
MM_LOENGLISH
MM_HIENGLISH
MM_TWIPS

When MM_ISOTROPIC mode is set, GDI makes adjustments to the window or viewport extents as necessary after calls to **SetWindowExt** or **SetViewportExt** to ensure that the x and y units remain the same size.

GetViewportExt (*hDC*) : *ptExtents*

Purpose This function retrieves the x and y extents of the display context's viewport.

Parameters *hDC* is a handle to a display context.

Return Value *ptExtents* is a long integer value containing the x and y extents in device units. The y extent is in the high-order word; the x extent is in the low-order word.

GetBrushOrg (*hDC*) : *ptOrigin*

Purpose This function retrieves the current brush origin for the given display context.

Parameter *hDC* is a handle to a display context.

Return Value *ptOrigin* is a long integer value containing the current origin of the brush. The x coordinate is in the low-order word and the y coordinate is in the high-order word. The coordinates are assumed to be in device units.

Notes The initial brush origin is at 0,0.

SetBrushOrg (*hDC*, *X*, *Y*) : *ptOldOrigin*

Purpose This function sets the origin of all brushes selected into the given display context.

Parameters *hDC* is a handle to a display context.

X and *Y* are short integers values specifying the origin in device coordinates.

Return Value *ptOldOrigin* is a long value containing the old origin of the brush. The x coordinate is in the low-order word and the y coordinate is in the high-order word.

Notes The initial brush origin is at 0,0.

SetBrushOrg should not be used with stock objects.

UnrealizeObject (*hBrush*) : *bUnrealized*

Purpose This function directs GDI to reset the origin of the given brush the next time it is selected.

Parameter *hBrush* is a handle to a logical brush.

<i>Return Value</i>	<i>bUnrealized</i> , a Boolean value, is nonzero if the function is successful, zero otherwise.
<i>Notes</i>	<p>UnrealizeObject should not be used with stock objects.</p> <p>This function must be called whenever a new brush origin is set (using SetBrushOrigin).</p> <p>The brush specified by <i>hBrush</i> must not be the currently selected brush of any display context.</p>

3.7 Clipping Region Functions

This section describes the functions used to examine and set the clipping region of a display context. The clipping region defines what portion of the display surface is clipped. Only images drawn inside the clipping region are copied to the surface. The default region is the whole surface.

GetClipBox (*hDC*, *lpRect*) : *nRgnType*

Purpose This function retrieves the dimensions of the tightest bounding rectangle around the current clipping boundary. The dimensions are copied to the buffer pointed to by *lpRect*.

Parameters *hDC* is a handle to a display context.

lpRect is a long pointer to the buffer to receive the rectangle dimensions. The buffer must have **RECT** data structure type.

Return Value *nRgnType* is a short integer flag specifying the clipping region's type. It can be any one of the following:

- ERROR
- NULLREGION
- SIMPLEREGION
- COMPLEXREGION

IntersectClipRect (*hDC*, *X1*, *Y1*, *X2*, *Y2*) : *nRgnType*

Purpose This function creates a new clipping region by forming the intersection of the current region and the rectangle specified by *X1*, *Y1*, *X2*, and *Y2*. GDI clips all subsequent output to fit within the new boundary.

Parameters *hDC* is a handle to a display context.

X1 and *Y1* are short integer values specifying the logical coordinates of the upper left corner of the rectangle.

X2 and *Y2* are short integer values specifying the logical coordinates of the lower right corner of the rectangle.

Return Value *nRgnType* is a short integer flag specifying the new clipping region's type. It can be any one of the following:

ERROR
NULLREGION
SIMPLEREGION
COMPLEXREGION

Notes The width of the rectangle specified by *X1*, *Y1*, *X2*, and *Y2* must not exceed 32,767 units.

OffsetClipRgn (*hDC*, *X*, *Y*) : *nRgnType*

Purpose This function moves the clipping region of the given device by the specified offsets. The function moves the region *X* units along the x-axis and *Y* units along the y-axis.

Parameters *hDC* is a handle to a display context.

X is a short integer value specifying the amount to move left or right, in logical units.

Y is a short integer value specifying the amount to move up or down, in logical units.

Return Value *nRgnType* is a short integer flag specifying the new region's type. It can be any one of the following:

ERROR
NULLREGION
SIMPLEREGION
COMPLEXREGION

ExcludeClipRect (*hDC*, *X1*, *Y1*, *X2*, *Y2*) : *nRgnType*

Purpose This function creates a new clipping region that consists of the existing clipping region less the specified rectangle.

Parameters *hDC* is a handle to a display context.

X1 and *Y1* are short integer values specifying the logical coordinates of the the upper left corner of the rectangle.

X2 and *Y2* are short integer values specifying the logical coordinates of the lower right corner of the rectangle.

Return Value *nRgnType* is a short integer flag specifying the new clipping region's type. It can be any one of the following:

ERROR
NULLREGION
SIMPLEREGION
COMPLEXREGION

Notes The width of the rectangle specified by *X1*, *Y1*, *X2*, and *Y2* must not exceed 32,767 units.

PtVisible (*hDC*, *X*, *Y*) : *bVisible*

Purpose This function indicates whether or not the given point is within the clipping region of the specified display context.

Parameters *hDC* is a handle to a display context.

X and *Y* are short integer values specifying the logical coordinates of the point to be checked.

Return Value *bVisible*, a Boolean value, is nonzero if the point is within the clipping region. Otherwise, it is zero.

RectVisible (*hDC*, *lpRect*) : *bVisible*

Purpose This function determines whether or not any part of the given rectangle lies within the clipping region of the specified display.

Parameters *hDC* is a handle to a display context.

lpRect is a long pointer to a data structure having **RECT** type.

Return Value *bVisible*, a Boolean value, is nonzero if some portion of the given rectangle lies within the clipping region. Otherwise, it is zero.

3.8 Region Functions

This section describes the utility functions used to create and modify regions.

**CombineRgn (*hDestRgn*, *hSrcRgn1*, *hSrcRgn2*, *nCombineMode*)
*: nRgnType***

Purpose This function creates a new region by combining two existing regions. The method used to combine the regions is specified by *nCombineMode*.

Parameters *hDestRgn* is a handle to the new region.

hSrcRgn1 is a handle to one existing region.

hSrcRgn2 is a handle to the other existing region.

nCombineMode is a short integer value specifying the operation to be performed on the two existing regions. It can be any one of the following:

Value	Operation
RGN_AND	Uses overlapping areas of both regions (intersection).
RGN_OR	Combines all of both regions (union).
RGN_XOR	Combines both regions but removes overlapping areas.
RGN_DIFF	Saves the areas of Region 1 (<i>hSrcRgn1</i>) that are not part of Region 2 (<i>hSrcRgn2</i>).
RGN_COPY	Creates a copy of Region 1 (<i>hSrcRgn1</i>).

Return Value *nRgnType* is a short integer flag value indicating the type of the resulting region. It can be any one of the following:

ERROR
NULLREGION
SIMPLEREGION
COMPLEXREGION

EqualRgn (*hSrcRgn1*, *hSrcRgn2*) : *bEqual*

Purpose This function checks the two given regions to determine if they are identical.

Parameters *hSrcRgn1* is a handle to one region.

hSrcRgn2 is a handle to the other region.

Return Value *bEqual*, a Boolean value, is nonzero if the two regions are equal. Otherwise, it is zero.

OffsetRgn (*hRgn*, *X*, *Y*) : *nRgnType*

Purpose This function moves the given region by the specified offsets. The function moves the region *X* units along the x-axis and *Y* units along the y-axis.

Parameters *hRgn* is a handle to a region.

X is a short integer value specifying the amount to move left or right.

Y is a short integer value specifying the amount to move up or down.

Return Value *nRgnType* is a short integer flag specifying the new region's type. It can be any one of the following:

ERROR
NULLREGION
SIMPLEREGION
COMPLEXREGION

CreateRectRgn (*X1*, *Y1*, *X2*, *Y2*) : *hRgn*

Purpose This function creates a rectangular region.

Parameters *X1* and *Y1* are short integer values specifying the upper left corner of the region.

X2 and *Y2* are short integer values specifying the lower right corner of the region.

Return Value If the function is successful, *hRgn* is a handle to a new region. Otherwise, it is NULL.

Notes The width of the rectangle specified by *X1*, *Y1*, *X2*, and *Y2* must not exceed 32,767 units.

CreateRectRgnIndirect (*lpRect*) : *hRgn*

Purpose This function creates a rectangular region.

Parameters *lpRect* is a long pointer to a data structure having **RECT** type. The structure must contain the coordinates of the upper left and lower right corners of the region.

Return Value If the function is successful, *hRgn* is a handle to a new region. Otherwise, it is NULL.

CreateEllipticRgn (*X1*, *Y1*, *X2*, *Y2*) : *hRgn*

Purpose This function creates an elliptical region.

Parameters *X1* and *Y1* are short integer values specifying the coordinates of the upper left corner of the bounding rectangle of the ellipse.

X2 and *Y2* are short integer values specifying the coordinates of the lower right corner of the bounding rectangle of the ellipse.

Return Value If the function is successful, *hRgn* is a handle to a new region. Otherwise, it is NULL.

Notes The width and height of the rectangle specified by *X1*, *Y1*, *X2*, and *Y2* must not exceed 32,767 units.

CreateEllipticRgnIndirect (*lpRect*) : *hRgn*

Purpose This function creates an elliptical region.

Parameters *lpRect* is a long pointer to a data structure having **RECT** type. The structure must contain the coordinates of the upper left and lower right corners of the bounding rectangle of the ellipse.

Return Value If the function is successful, *hRgn* is a handle to a new region. Otherwise, it is NULL.

CreatePolygonRgn (*lpPoints*, *nCount*, *nPolyFillMode*) : *hRgn*

Purpose This function creates a polygonal region.

Parameters *lpPoints* is a long pointer to an array of **POINT** data structures. Each point specifies the coordinates of one vertex of the polygon.

nCount is a short integer value specifying the number of points in the array.

nPolyFillMode is a short integer value specifying the polygon filling mode to be used for filling the region. It can be ALTERNATE or WINDING (see the **SetPolyFillMode** function for an explanation of these modes).

Return Value If the function is successful, *hRgn* is a handle to a new region. Otherwise, it is NULL.

PtInRegion (*hRgn*, *X*, *Y*) : *bSuccess*

Purpose This routine specifies whether or not the point given by *X* and *Y* is in the given region.

Parameters *hRgn* is a handle to a region.

X is a short integer value specifying the x coordinate (in logical coordinates) of the point.

Y is a short integer value specifying the y coordinate (in logical coordinates) of the point.

Return Value *bSuccess*, a Boolean value, is nonzero if the point is in the region. Otherwise, it is zero.

3.9 Text Justification Functions

This section describes the functions used to justify text to be written to the display surface. The functions are intended to be used with the **TextOut** function which carries out the actual output of text.

SetTextJustification (*hDC*, *nBreakExtra*, *nBreakCount*) : *nSet*

Purpose This function prepares GDI to justify a line of text using the justification parameters specified by *nBreakExtra* and *nBreakCount*. To justify text, GDI distributes extra pixels among break characters in a text line written by the **TextOut** function. The break character, used to delimit words, is usually the space character (ASCII 32), but may be defined as some other character by a font. (**GetTextMetric** can be used to retrieve a font's break character).

The **SetTextJustification** function prepares the justification by defining the amount of space to be added. The *nBreakExtra* parameter specifies the total amount of space (in logical units) to add to the line. The *nBreakCount*

parameter specifies how many break characters are in the line. The subsequent **TextOut** function distributes the extra space evenly between each break character in the line.

The **GetTextExtent** function is always used with **SetTextJustification**. **GetTextExtent** computes the width before justification of a given line. This width must be known before an appropriate *nBreakExtra* value can be computed.

SetTextJustification can be used to justify a line containing multiple runs in different fonts. In this case, the line must be created piecemeal by justifying and writing each run separately.

Because rounding errors can occur during justification, GDI keeps a running error term that defines the current error. When justifying a line containing multiple runs, **GetTextExtent** automatically uses this error term when it computes the extent of the next run, allowing **TextOut** to blend the error into the new run. After each line has been justified, this error term must be cleared to prevent it from being incorporated into the next line. The term can be cleared by calling **SetTextJustification** with *nBreakExtra* set to zero.

Parameters *hDC* is a handle to a display context.

nBreakExtra is a short integer value specifying the total extra space (in logical units) to be added to the line of text. If the current mapping mode is not MM_TEXT, *nBreakExtra* is transformed and rounded to the nearest pixel.

nBreakCount is a short integer number specifying the number of break characters in the line.

Return Value *nSet*, a short integer value, is 1 if the function is successful. Otherwise, it is 0.

GetTextExtent (*hDC*, *lpString*, *nCount*) : *ptTextExtents*

Purpose This function computes the width and height of the line of text pointed to by *lpString*. **GetTextExtent** uses the currently selected font to compute the dimensions of the string. The width and height, specified in logical units, are computed without considering the current clipping region.

<i>Parameters</i>	<i>hDC</i> is a handle to a display context. <i>lpString</i> is a long pointer to a text string. <i>nCount</i> is a short integer value specifying the number of characters in the text string.
<i>Return Value</i>	<i>ptTextExtents</i> , a long integer value, contains the dimensions of the string. The height is in the high-order word; the width is in the low-order word.
<i>Note</i>	Since some devices do not place characters in regular cell arrays (e.g. kerning), the sum of the extents of the characters in a string may not be equal to the extent of the string.

SetTextCharacterExtra (*hDC*, *nCharExtra*) : *nOldCharExtra*

<i>Purpose</i>	This function sets the amount of intercharacter spacing. GDI adds this intercharacter spacing to each character, including break characters, when it writes a line of text to the display surface.
<i>Parameters</i>	<i>hDC</i> is a handle to a display context. <i>nCharExtra</i> is a short integer value specifying the amount of extra space (in logical units) to be added to each character. If the current mapping mode is not MM_TEXT, <i>nCharExtra</i> is transformed and rounded to the nearest pixel.
<i>Return Value</i>	<i>nOldCharExtra</i> is a short integer value specifying the previous intercharacter spacing.

GetTextCharacterExtra (*hDC*) : *nCharExtra*

<i>Purpose</i>	This function retrieves the current intercharacter spacing. The intercharacter spacing defines the extra space (in logical units) that TextOut adds to each character as it writes a line. The spacing is used to expand lines of text. If the current mapping mode is not MM_TEXT, GetTextCharExtra transforms and rounds the result to the nearest unit.
<i>Parameters</i>	<i>hDC</i> is a handle to a display context.
<i>Return Value</i>	<i>nCharExtra</i> is a short integer value specifying the current intercharacter spacing.

3.10 Metafile Functions

This section describes the functions used to create, manipulate, and delete metafiles. A metafile can be created using the **CreateMetaFile** function. Once created, an application can draw on the file using the functions listed below. When drawing is complete, the application must close the metafile by using the **CloseMetaFile** function. This function creates a metafile handle that can be used to access and play the metafile. Table 3.2 lists the GDI functions which can be used in a metafile.

Table 3.2
GDI Functions and Metafiles

Arc	PaintRgn	SetMapMode
BitBlt	PatBlt	SetPixel
Ellipse	Pie	SetPolyFillMode
Escape	Polygon	SetROP2
ExcludeClipRect	Polyline	SetRelAbs
FillRgn	Rectangle	SetStretchBltMode
FloodFill	RestoreDC	SetTextCharExtra
FrameRgn	RoundRect	SetTextColor
IntersectClipRect	SaveDC	SetTextJustification
InvertRgn	ScaleViewportExt	SetViewportExt
LineTo	ScaleWindowExt	SetViewportOrg
MoveTo	SelectClipRgn	SetWindowExt
OffsetClipRect	SelectObject	SetWindowOrg
OffsetViewportOrg	SetBkColor	StretchBlt
OffsetWindowOrg	SetBkMode	TextOut

SaveDC and **RestoreDC** calls should be exactly balanced within a metafile; that is, there should be as many calls to **SaveDC** as to **RestoreDC**. Each **RestoreDC** call must restore a display context that was saved within the metafile, that is, relative restore levels should be used (for example, "RestoreDC(hDC, -1)").

BitBlt and **StretchBlt** copy source bitmaps to the file. The metafile display context can be used as both source and destination, but is extremely expensive in terms of memory and should be used with caution.

SelectObject copies the logical object to the file. On playback, GDI creates and selects the actual object. At the end of playback, GDI deletes the object.

Escape always passes a NULL value as the long pointer to output data.

CreateMetaFile (*lpFilename*) : *hDC*

Purpose This function creates a metafile display context.

Parameters *lpFilename* is a long pointer to a null-terminated character string specifying the name of the metafile. If *lpFilename* is NULL, a display context for a memory metafile is returned.

Return Value If the function is successful, *hDC* is a handle to a metafile display context. Otherwise, it is NULL.

CloseMetaFile (*hDC*) : *hMF*

Purpose This function closes the metafile DC and creates a metafile handle that can be used to play the metafile using **PlayMetaFile**.

Parameters *hDC* is a handle to the metafile display context to be closed.

Return Value If the function is successful, *hMF* is a handle to the metafile. Otherwise, it is NULL.

GetMetaFile (*lpFilename*) : *hMF*

Purpose This function creates a handle for the metafile named by *lpFilename*.

Parameters *lpFilename* is a long pointer to a null-terminated ASCII string specifying the MS-DOS filename of the metafile. The metafile is assumed to exist.

Return Value If the function is successful, *hMF* is a handle to a metafile. Otherwise, it is NULL.

CopyMetaFile (*hSrcMetaFile*, *lpFilename*) : *hMF*

Purpose This function copies the source metafile to the file named by *lpFilename* and returns a handle to the new metafile. If *lpFilename* is NULL, the source is copied to a memory metafile.

Parameters *hSrcMetaFile* is a handle to a source metafile.

lpFilename is a long pointer to a null-terminated character string specifying the file to receive the metafile.

Return Value *hMF* is a handle to the new metafile.

PlayMetaFile (*hDC*, *hMF*) : *bPlayed*

Purpose This function plays the contents of the specified metafile on the given device context. The metafile can be played any number of times.

Parameters *hDC* is a handle to a display context.

hMF is a handle to a metafile.

Return Value *bPlayed*, a Boolean value, is nonzero if the function is successful. Otherwise, it is zero.

DeleteMetaFile (*hMF*) : *bFreed*

Purpose This function deletes access to a metafile by freeing the system resources associated with that metafile. It does not destroy the metafile itself, but it invalidates the metafile handle *hMF*. The metafile can be used again by getting a new handle using the **GetMetaFile** function.

Parameters *hMF* is a handle to a metafile.

Return Value *bFreed*, a Boolean value, is nonzero if the metafile's system resources are deleted. It is zero if *hMF* is not a valid handle.

GetMetaFileBits (*hMF*) : *hMem*

Purpose This function returns a handle to a global memory block containing the specified metafile as a collection of bits. The memory block can be used to determine the size of the metafile or to save the metafile as a file. The memory block should not be modified.

Parameters *hMF* is a handle to a memory metafile.

<i>Return Value</i>	<i>hMem</i> is a handle to the metafile in global memory, if the function is successful. If an error occurs, <i>hMem</i> is NULL.
<i>Notes</i>	The handle used as the <i>hMF</i> parameter becomes invalid when GetMetaFileBits returns, so <i>hMem</i> must be used to refer to the metafile.
	Memory blocks created by this function are unique to the calling application and are not shared by other applications. These blocks are automatically deleted when the application terminates.

SetMetaFileBits (*hMem*) : *hMF*

<i>Purpose</i>	This function creates a memory metafile from the data in the global memory block specified by <i>hMem</i> .
<i>Parameters</i>	<i>hMem</i> is a handle to a global memory block containing metafile data. It is assumed that the data was previously created using the GetMetaFileBits function.
<i>Return Value</i>	<i>hMF</i> is a handle to a memory metafile if the function is successful. Otherwise, <i>hMF</i> is NULL.
<i>Notes</i>	After SetMetaFileBits returns, <i>hMF</i> should be used to refer to the metafile instead of <i>hMem</i> .

3.11 Control Functions

This section describes the functions used to control a display device.

Escape (*hDC*, *nEscape*, *nCount*, *lpInData*, *lpOutData*) : *nResult*

<i>Purpose</i>	This function allows applications to access facilities of a particular device that are not directly available through GDI. Escape calls made by an application are translated and sent to the device driver.
<i>Parameters</i>	<i>hDC</i> is a handle to a display context. <i>nEscape</i> is a short integer value specifying the escape function to be performed. The following predefined functions are available:

STARTDOC
ENDDOC
NEWFRAME
NEXTBAND
SETABORTPROC
ABORTDOC
DRAFTMODE
SETCOLORTABLE
GETCOLORTABLE
GETPHYSPAGESIZE
GETPRINTINGOFFSET
GETSCALINGFACTOR
FLUSHOUTPUT
QUERYESCSUPPORT

Devices can define additional escape functions.

nCount is a short integer number specifying the number of bytes of data pointed to by *lpInData*.

lpInData is a long pointer to the input data structure required for this escape.

lpOutData is a long pointer to the data structure to receive output from this escape. If *lpOutData* is NULL, no data is returned.

Return Value *nResult*, a short integer value, is positive if the escape function was successful (or is implemented, for the QUERYESCSUPPORT escape). *nResult* is zero if the escape is not implemented. A negative value indicates an error.

The following descriptions give the specific syntax and meaning of each **Escape** call.

Escape (*hDC*, *STARTDOC*, *nCount*, *lpInData*, *lpOutData*) : *nResult*

Purpose This escape informs the device driver that a new print job is starting and that all subsequent NEWFRAME calls should be spooled under the same job, until an ENDDOC call occurs. This ensures that documents longer than one page will not be interspersed with other jobs.

Parameters *nCount* is a short integer value specifying the number of characters in the string pointed to by *lpInData*.

lpInData is a long pointer to a null-terminated string specifying the name of the document. The document name is displayed in the spooler window.

lpOutData is not used and can be set to NULL.

Return Value *nResult*, a short integer value, is -1 if an error occurs, such as insufficient memory or an invalid port specification. Otherwise, it is positive.

Notes The correct sequence of events in a printing operation are as follows:

1. Create printer display context.
2. Set the abort function to keep out of disk space errors from aborting a printing operation. An abort procedure that handles these errors must be set using the SETABORTPROC escape.
3. Begin the printing operation with STARTDOC.
4. Begin each new page with NEWFRAME, or each new band with NEXTBAND.
5. End the printing operation with ENDDOC.

On a printing error, the ENDDOC escape should not be used to terminate the printing operation, and the ABORTDOC escape should be used to terminate a next banding operation only.

Escape (*hDC*, ENDDOC, *nCount*, *lpInData*, *lpOutData*) : *nResult*

Purpose This escape ends a print job started by a STARTDOC escape.

Parameters *nCount* is not used and can be set to NULL.

lpInData is not used and can be set to NULL.

lpOutData is not used and can be set to NULL.

Return Value *nResult*, a short integer value, is positive if the function is successful. Otherwise, it is negative.

Notes On a printing error, the ENDDOC escape should not be used to terminate the printing operation, and the ABORTDOC escape should be used to terminate a next banding operation only.

Escape (*hDC*, *NEWFRAME*, *nCount*, *lpInData*, *lpOutData*) : *nResult*

Purpose This escape informs the device that the application has finished writing to a page. This escape is typically used with a printer to direct the device driver to advance to a new page.

Parameters *nCount* is not used and can be set to NULL.

lpInData is not used and can be set to NULL.

lpOutData is not used and can be set to NULL.

Return Value *nResult*, a short integer value, is positive if the escape is successful. Otherwise, *nResult* is one of the following:

Error Code	Meaning
SP_ERROR	General error.
SP_APPABORT	The job was aborted because the application's abort function returned zero.
SP_USERABORT	The user aborted the job through the spooler task.
SP_OUTOFDISK	Not enough disk space is currently available for spooling, and no more space will become available.
SP_OUTOFMEMORY	Not enough memory is available for spooling.

The application can determine whether the error has been reported to the user by testing the SP_NOTREPORTED bit of the returned error code. If the SP_NOTREPORTED bit is set, the error has not been reported. If the bit is zero, the user has been notified of the error.

Escape (*hDC*, *NEXTBAND*, *nCount*, *lpInData*, *lpOutData*) : *nResult*

Purpose This escape informs the device driver that the application has finished writing to a band, causing the device driver to send the band to the spooler and return the coordinates of the next band. This escape is used by applications that handle banding themselves.

- Parameters* *nCount* is not used and can be set to NULL.
lpInData is not used and can be set to NULL.
lpOutData is a long pointer to a **RECT** data structure. The device driver copies the device coordinates of the next band into this structure.
- Return Value* *nResult*, a short integer value, is a positive if the escape is successful. Otherwise, *nResult* is one of the following:

Error Code	Meaning
SP_ERROR	General error.
SP_APPABORT	The job was aborted because the application's abort function returned zero.
SP_USERABORT	The user aborted the job through the spooler task.
SP_OUTOFDISK	Not enough disk space is currently available for spooling, and no more space will become available.
SP_OUTOFMEMORY	Not enough memory is available for spooling.

The application can determine whether the error has been reported to the user by testing the SP_NOTREPORTED bit of the returned error code. If the SP_NOTREPORTED bit is set, the error has not been reported. If the bit is zero, the user has been notified of the error.

Escape (*hDC*, *SETABORTPROC*, *nCount*, *lpInData*, *lpOutData*) : *nResult*

- Purpose* This escape sets the abort function for the print job. If an application wants to allow the print job to be cancelled during spooling, it must set the abort function before the print job is started with the STARTDOC escape. The spooler calls the abort function during spooling to allow the application to cancel the print job or to handle out-of-disk-space conditions. If no abort function is set, the print job will fail if there is not enough disk space for spooling.

Parameters *nCount* is not used and can be set to NULL.
lpInData is a long pointer to the abort function.
lpOutData is not used and can be set to NULL.

Return Value *nResult*, a short integer value, is positive if the function is successful. Otherwise, it is negative.

Notes The abort function has the following form:

AbortFunc (hPr, code) : bContinue

hPr is a handle to the printer display context.

code, a short integer value, is 0 if no error has occurred. It is SP_OUTOFDISK if the spooler is currently out of disk space, but more will become available if the application is willing to wait.

bContinue, a Boolean value, should be nonzero if the print job is to be continued, and zero if it is cancelled.

Escape (hDC, ABORTDOC, nCount, lpInData, lpOutData) : nResult

Purpose This escape aborts the current job, erasing everything the application has written to the device since the last END-DOC escape.

The ABORTDOC escape should be used for printing operations that do not specify an abort function (SETABORT-PROC escape), and to terminate printing operations that have not yet reached their first NEWFRAME or NEXTBAND call.

Parameters *nCount* is not used and can be set to NULL.
lpInData is not used and can be set to NULL.
lpOutData is not used and can be set to NULL.

Return Value *nResult*, a short integer value, is positive if the function is successful. Otherwise, it is negative.

Notes On a printing error, the ENDDOC escape should not be used to terminate the printing operation, and the ABORTDOC escape should be used to terminate a next banding operation only.

Escape (*hDC*, *DRAFTMODE*, *nCount*, *lpInData*, *lpOutData*) : *nResult*

Purpose This escape turns draft mode off or on. Turning draft mode on instructs the device driver to print faster and with lower quality (if necessary). The draft mode can only be changed at page boundaries (for example, after a NEWFRAME escape.)

Parameters *nCount* is a short integer value specifying the number of bytes pointed to by *lpInData*.

lpInData is a long pointer to a short integer value specifying the draft mode: 1 for draft mode on, 0 for off.

lpOutData is not used and can be set to NULL.

Return Value *nResult*, a short integer value, is positive if the function is successful. Otherwise, it is negative.

Notes The default draft mode is off.

Escape (*hDC*, *SETCOLORTABLE*, *nCount*, *lpInData*, *lpOutData*) : *nResult*

Purpose This escape sets an RGB color table entry. If the device cannot supply the exact color, the function sets the entry to the closest possible approximation of the color.

Parameters *nCount* is not used and can be set to NULL.

lpInData is a long pointer to a data structure having the following items:

```
WORD Index;
LONG rgb;
```

The *Index* is the color table index. Color table entries start at 0 for the first entry. The *rgb* is the desired RGB color value.

lpOutData is a long pointer to an RGB color value. The device driver copies the new color value to this location.

Return Value *nResult*, a short integer value, is positive if the function is successful. Otherwise, it is negative.

Notes A device's color table is a shared resource; changing the system display color for one window changes it for all windows.

The SETCOLORTABLE escape has no effect on devices with fixed color tables.

Escape (*hDC*, *GETCOLORTABLE*, *nCount*, *lpInData*, *lpOutData*)
: *nResult*

Purpose This escape retrieves an RGB color color table entry and copies it to the location specified by *lpOutData*.

Parameters *nCount* is not used and can be set to NULL.

lpInData is a long pointer to a short integer value specifying the index of a color table entry. Color table indexes start at 0 for the first table entry.

lpOutData is a long pointer to an RGB color value.

Return Value *nResult*, a short integer value, is positive if the function is successful. Otherwise, it is negative.

Escape (*hDC*, *GETPHYSAGESIZE*, *nCount*, *lpInData*, *lpOutData*)
: *nResult*

Purpose This escape retrieves the physical page size and copies it to the location pointed to by *lpOutData*.

Parameters *nCount* is not used and can be set to NULL.

lpInData is not used and can be set to NULL.

lpOutData is a long pointer to a **POINT** structure. The device driver fills this structure with the physical page dimensions in device units. The horizontal size is in the x coordinate; the vertical size is in the y coordinate.

Return Value *nResult*, a short integer value, is positive if the function is successful. Otherwise, it is negative.

Escape (*hDC*, *GETPRINTINGOFFSET*, *nCount*, *lpInData*, *lpOutData*)
: *nResult*

Purpose This escape retrieves the offset, from the upper left hand corner of the physical page, where the actual printing or drawing begins. This escape function is not generally useful for devices that allow the user to set the printable origin by hand.

Parameters *nCount* is not used and can be set to NULL.

lpInData is not used and can be set to NULL.

lpOutData is a long pointer to a **POINT** structure. The device driver fills this structure with the offset in device units.

Return Value *nResult*, a short integer value, is positive if the function is successful. Otherwise, it is negative.

Escape (*hDC*, *GETSCALINGFACTOR*, *nCount*, *lpInData*, *lpOutData*) : *nResult*

Purpose This escape retrieves the scaling factors for the x and y axes of a printing device. For each scaling factor, the escape copies an exponent of two to the location pointed to by *lpOutData*. For example, the value 3 is copied to *lpOutData* if a scaling factor is 8.

Scaling factors are used by printing devices that cannot support graphics at the same resolution as the device resolution.

Parameters *nCount* is not used and can be set to NULL.

lpInData is not used and can be set to NULL.

lpOutData is a long pointer to a **POINT** structure. The device driver copies the scaling factors for the x and y axes to the structure.

Return Value *nResult*, a short integer value, is positive if the function is successful. Otherwise, it is negative.

Escape (*hDC*, *FLUSHOUTPUT*, *nCount*, *lpInData*, *lpOutData*) : *nResult*

Purpose This escape flushes output in the device's buffer.

Parameters *nCount* is not used and can be set to NULL.

lpInData is not used and can be set to NULL.

lpOutData is not used and can be set to NULL.

Return Value *nResult*, a short integer value, is positive if the function is successful. Otherwise, it is negative.

Escape (*hDC*, *QUERYYESCSUPPORT*, *nCount*, *lpInData*, *lpOutData*) : *nResult*

Purpose This escape finds out whether a particular escape function is implemented by the device driver. The return value is nonzero for implemented escape functions, and zero for unimplemented escape functions.

Parameters *nCount* is a short integer value specifying the number of bytes pointed to by *lpInData*.

lpInData is a long pointer to a short integer value specifying the escape function to be checked.

lpOutData is not used and can be set to NULL.

Return Value *nResult*, a short integer value, is nonzero for implemented escape functions. Otherwise, it is zero.

3.12 GDI Information Functions

This section describes the functions used to retrieve information about GDI and specific devices.

EnumFonts (*hDC*, *lpFacename*, *lpFontFunc*, *lpData*) : *nResult*

Purpose This function enumerates the fonts available on a given device. For each font having the facename specified by *lpFacename*, **EnumFonts** retrieves information about that font and passes it to the function pointed to by *lpFontFunc*. The callback function, an application-supplied function, can process the font information as desired. Enumeration continues until there are no more fonts or the callback function returns zero.

Parameters *hDC* is a handle to a display context.

lpFacename is a long pointer to a null-terminated ASCII string specifying the facename of the desired fonts. If *lpFacename* is NULL, **EnumFonts** randomly selects and enumerates one font of each face available.

lpFontFunc is a long pointer to the application-supplied callback function.

lpData is a long pointer to the application-supplied data. The data is passed to the callback function along with the font information.

Return Value *nResult*, a short integer value, is equal to the last value returned by the callback function. Its meaning is user-defined.

Notes

The callback function must have the form:

lpFontFunc(lpLogFont, lpTextMetrics, FontType, lpData):nResult

where *lpLogFont* is a long pointer to a data structure having **LOGFONT** type, *lpTextMetrics* is a long pointer to a data structure having **TEXTMETRIC** type, *FontType* is a short integer value specifying the type of the font, and *lpData* is a long pointer to the application-supplied data passed to **EnumFonts**.

The return value, *nResult*, can be any integer value.

The AND operator (*&*) can be used with the **RASTER_FONTTYPE** and **DEVICE_FONTTYPE** constants to determine the font type. The **RASTER_FONTTYPE** bit of *FontType* specifies whether or not the font is a raster or vector font. If the bit is 1, the font is a raster font; if 0, a vector font. The **DEVICE_FONTTYPE** bit of *FontType* specifies whether or not the font is a device- or GDI-based font. If the bit is 1, the font is a device-based font; if 0, a GDI-based font.

If the device is capable of text transformations (scaling, italicizing, etc.) only the base font will be enumerated. The user is responsible for inquiring the device's text transformation abilities to determine which additional fonts are available directly from the device. GDI can simulate the emboldened, italic, underlined, and strikeout attributes for any font.

EnumFonts only enumerates fonts from the GDI internal table. This does not include fonts that are generated by a device, such as fonts that are transformations of fonts from the internal table. **GetDeviceCaps** can be used to determine what type of transformations a device can perform. This information is available using the **TEXTCAPS** index.

GDI can scale GDI-based raster fonts by 1 to 5 horizontally and 1 to 8 vertically, unless **PROOF_QUALITY** is being used.

EnumObjects (*hDC, nObjectType, lpObjectFunc, lpData*) : *nResult*

Purpose

This function enumerates the pens and brushes available on a device. For each object belonging to the given style, the callback function is called with the information for that object. The callback function is called until there are no more objects or the callback function returns zero.

Parameters *hDC* is a handle to a display context.
nObjectType is a short integer value identifying the object type. It can be one of the following:

OBJ_PEN
OBJ_BRUSH

lpObjectFunc is a long pointer to the application-supplied callback function.

lpData is a long pointer to the application-supplied data. The data is passed to the callback function along with the object information.

Return Value *nResult*, a short integer value, is equal to the last value returned by the callback function. Its meaning is user-defined.

Notes The callback function must have the form:

functionname (lpObj, lpData)

where *lpObj* is a pointer to a logical pen or brush and *lpData* is a long pointer to the application-supplied data passed to **EnumObjects**.

GetTextFace (*hDC, nCount, lpFacename*) : *nCopied*

Purpose This function copies the facename of the currently selected font into a buffer pointed to by *lpFacename*. The facename is copied as a null-terminated ASCII string. The *nCount* parameter specifies the maximum number of characters to copy. If the name is longer than *nCount*, is it truncated.

Parameters *hDC* is a handle to a display context.

nCount is a short integer value specifying the size of the buffer in bytes.

lpFacename is a long pointer to the buffer to receive the facename.

Return Value *nCopied*, a short integer value, is the actual number of bytes copied to the buffer. It is 0 if there is an error.

GetTextMetrics (*hDC, lpMetrics*) : *bRetrieved*

Purpose This function fills the buffer pointed to by *lpMetrics* with the metrics for the currently selected font.

Parameters *hDC* is a handle to a display context.

lpMetrics is a long pointer to a data structure having **TEXTMETRIC** type.

Return Value *bRetrieved*, a Boolean value, is nonzero if the function is successful. Otherwise, it is zero.

GetDeviceCaps (*hDC, nIndex*) : *nValue*

Purpose This function retrieves device-specific information about a given display device. The *nIndex* parameter specifies the type of information desired.

Parameters *hDC* is a handle to a display context.

nIndex is a short integer value specifying the item to return. It can be any one of the values given in Table 3.3.

Return Value *nValue* is the integer value of the desired item.

Table 3.3
GDI Information Indexes

Index	Meaning
DRIVERVERSION	The version number. (100H)
TECHNOLOGY	The device technology. It can be any one of the following: 0 Vector plotter 1 Raster display 2 Raster printer 3 Raster camera 4 Character-stream, PLP 5 Metafile, VDM 6 Display-file
HORZSIZE	The width of the physical display in millimeters.
VERTSIZE	The height of the physical display in millimeters.
HORZRES	The width of the display in pixels.

Table 3.3 (continued)

Index	Meaning
VERTRES	The height of the display in raster lines.
BITSPIXEL	The number of adjacent color bits for each pixel.
PLANES	The number of color planes.
NUMBRUSHES	The number of device-specific brushes.
NUMPENS	The number of device-specific pens.
NUMFONTS	The number of device-specific fonts.
NUMCOLORS	The number of entries in the device's color table.
ASPECTX	The relative width of a device pixel as used for line drawing.
ASPECTY	The relative height of a device pixel as used for line drawing.
ASPECTXY	The diagonal width of the device pixel as used for line drawing.
PDEVICESIZE	The size of the internal data structure PDEVICE .
CLIPCAPS	A flag indicating the clipping capabilities of the device. It is 1 if the device can clip to a rectangle. It is 0 if it cannot clip.
RASTERCAPS	A flag indicating the raster capabilities of the device, as follows: RC_BITBLT - Capable of transferring bitmaps. RC_BANDING - Requires banding support. RC_SCALING - Capable of scaling.
CURVECAPS	A bitmask indicating the curve capabilities of the device. The bits have the following meanings: bit 0 - set means can do circles bit 1 - set means can do pie wedges bit 2 - set means can do chord arcs bit 3 - set means can do ellipses bit 4 - set means can do wide borders bit 5 - set means can do styled borders bit 6 - set means can do borders that are wide and styled bit 7 - set means can do interiors The high-order byte is 0.

Table 3.3 (continued)

Index	Meaning
LINECAPS	<p>A bitmask indicating the line capabilities of the device. The bits have the following meanings: bit 0 - reserved bit 1 - set means can do polyline bit 2 - reserved bit 3 - reserved bit 4 - set means can do wide lines bit 5 - set means can do styled lines bit 6 - set means can do lines that are wide and styled bit 7 - set means can do interiors The high-order byte is 0.</p>
POLYGONALCAPS	<p>A bitmask indicating the polygonal capabilities of the device. The bits have the following meanings: bit 0 - set means can do alternate fill polygon bit 1 - set means can do rectangle bit 2 - set means can do winding number fill polygon bit 3 - set means can do scanline bit 4 - set means can do wide borders bit 5 - set means can do styled borders bit 6 - set means can do borders that are wide and styled bit 7 - set means can do interiors The high-order byte is 0.</p>
TEXTCAPS	<p>A bitmask indicating the text capabilities of the device. The bits have the following meanings: bit 0 - set means can do character output precision bit 1 - set means can do stroke output precision bit 2 - set means can do stroke clip precision bit 3 - set means can do 90 degree character rotation bit 4 - set means can do any character rotation bit 5 - set means can do scaling independent of X and Y bit 6 - set means can do doubled character for scaling bit 7 - set means can do integer multiples for scaling bit 8 - set means can do any multiples for exact scaling bit 9 - set means can do double weight characters bit 10 - set means can do italicizing bit 11 - set means can do underlining bit 12 - set means can do strikeouts bit 13 - set means can do raster fonts bit 14 - set means can do vector fonts bit 15 - Reserved. Must be returned zero.</p>
<i>Note</i>	<p>All of the available abilities are described under the LOGFONT data structure.</p>

SetEnvironment (*lpPortName*, *lpEnviron*, *nCount*) : *nCopied*

<i>Purpose</i>	This function copies the contents of the buffer specified by <i>lpEnviron</i> into the environment associated with the device attached to the system port specified by <i>lpPortName</i> . SetEnvironment overwrites any existing environment. If there is no environment for the given port, SetEnvironment creates it. If <i>nCount</i> is 0, the existing environment is deleted and not replaced.
<i>Parameters</i>	<i>lpPortName</i> is a long pointer to a null-terminated string specifying the name of the desired port. <i>lpEnviron</i> is a long pointer to the buffer containing the new environment. <i>nCount</i> is a short integer value specifying the number of bytes to be copied.
<i>Return Value</i>	<i>nCopied</i> is a short integer value specifying the number of bytes copied to the environment. It is 0 if there is an error. It is -1 if the environment is deleted.
<i>Notes</i>	The first field in the buffer pointed to by <i>lpEnviron</i> must be the same as passed in the <i>lpDeviceName</i> parameter of the CreateDC function. If <i>lpPortName</i> specifies a null port (as defined in the WIN.INI file), the device name pointed to by <i>lpEnviron</i> is used to locate the desired environment.

GetEnvironment (*lpPortName*, *lpEnviron*, *nMaxCount*) : *nCopied*

<i>Purpose</i>	This function copies the current environment associated with the device attached to the system port specified by <i>lpPortName</i> into the buffer specified by <i>lpEnviron</i> . The environment, maintained by GDI, contains binary data used by GDI whenever a display context (DC) is created for the device on the given port.
	The function fails if there is no environment for the given port.
<i>Parameters</i>	<i>lpPortName</i> is a long pointer to a null-terminated string specifying the name of the desired port. <i>lpEnviron</i> is a long pointer to the buffer that will receive the environment. <i>nMaxCount</i> is a short integer value specifying the maximum number of bytes to copy.

Return Value *nCopied* is a short integer value specifying the number of bytes copied to *lpData*. It is 0 if the environment cannot be found.

Notes The first field in the buffer pointed to by *lpEnviron* must be the same as passed in *lpDeviceName* parameter of the **CreateDC** function. If *lpPortName* specifies a null port (as defined in in the WIN.INI file), the device name pointed by *lpEnviron* is used to locate the desired environment.

GetDCOrg (*hDC*) : *ptOrigin*

Purpose This function returns the final translation origin for the screen display context. The final translation origin specifies the offset used by Windows to translate device coordinates into client coordinates for points in an application's window. The final translation origin is relative to the physical origin of the screen display.

Parameters *hDC* is a handle to a display context.

Return Value *ptOrigin* is long integer value containing the final translation origin in device coordinates. The y coordinate is in the high-order word; the x coordinate in the low-order word.

Notes The final translation origin is nonzero for screen display contexts only.

GetNearestColor (*hDC*, *rgbColor*) : *rgbSolidColor*

Purpose This function takes a logical color and returns the closest logical color the device can represent.

Parameters *hDC* is a handle to a device display context.

rgbColor is an RGB color value specifying the color to be matched.

Return Value *rgbSolidColor*, an RGB color value, is the solid color closest to *rgbColor* that the device can represent.

3.13 Conversion Functions

This section describes the functions used to convert logical coordinates into device coordinates, and vice versa. Logical coordinates specify points in GDI's logical coordinate system and are device-independent. Device coordinates specify points on a device's display surface and are device-dependent.

DPtolP (*hDC, lpPoints, nCount*) : *bConverted*

Purpose This function converts device points into logical points. The function maps the coordinates of each point specified by *lpPoints* from the device coordinate system into GDI's logical coordinate system. The conversion depends on the current mapping mode and the settings of the origins and extents for the device's window and viewport.

Parameters *hDC* is a handle to a display context.

lpPoints is a long pointer to an array of points. Each point must be a data structure having **POINT** type.

nCount is a short integer value specifying the number of points pointed to by *lpPoints*.

Return Value *bConverted*, a Boolean value, is nonzero if all points are converted. Otherwise, it is zero.

LPtolDP (*hDC, lpPoints, nCount*) : *bConverted*

Purpose This function converts logical points into device points. The function maps the coordinates of each point specified by *lpPoints* from GDI's logical coordinate system into a device coordinate system. The conversion depends on the current mapping mode.

Parameters *hDC* is a handle to a display context.

lpPoints is a long pointer to an array of points. Each point must be a data structure having **POINT** type.

nCount is a short integer value specifying the number of points pointed to by *lpPoints*.

Return Value *bConverted*, a Boolean value, is nonzero if all points are converted. Otherwise, it is zero.

Chapter 4

System Resource Functions

4.1	Introduction	191
4.2	Module Manager Functions	191
4.3	Memory Manager Functions	196
4.4	Task Functions	206
4.5	Resource Manager Functions	207
4.6	String Translation Functions	217
4.7	Atom Manager Functions	219
4.8	Windows Initialization File Functions	222
4.9	Debugging Function	225
4.10	Communication Functions	225
4.11	Sound Functions	235
4.12	Utility Functions	242
4.13	File I/O Functions	245

(

)

)

4.1 Introduction

This chapter describes the system resource functions. These functions provide access to the resources of the Windows system, such as memory and file resources.

4.2 Module Manager Functions

This section describes the functions used to access the code and data in Windows executable modules.

GetModuleHandle (*lpModuleName*) : *hModule*

Purpose This function retrieves the module handle of the specified module.

Parameter *lpModuleName* is a long pointer to a null-terminated ASCII string specifying the module name.

Return Value *hModule* is a handle to the module if the function is successful. Otherwise, it is NULL.

GetModuleUsage (*hModule*) : *nCount*

Purpose This function returns the reference count of a given module.

Parameters *hModule* is a module handle or an instance handle.

Return Value *nCount* is a short integer value specifying the reference count of the module.

GetModuleFileName (*hModule*, *lpFilename*, *nSize*) : *nLength*

Purpose This function retrieves the name of the executable file from which the specified module was loaded. The function copies the null-terminated name into the buffer pointed to by *lpFilename*.

Parameters *hModule* is a module handle or an instance handle.

lpFilename is a long pointer to the buffer where the filename will be stored.

nSize is a short integer specifying the size of the buffer. If the filename is longer than *nSize*, it is truncated.

Return Value *nLength* is a short integer specifying the actual length of the string stored in the buffer.

GetInstanceData (*hInstance*, *pData*, *nCount*) : *nBytes*

Purpose This function copies data from a previous instance of an application into the data area of the current instance. The *hInstance* parameter specifies which instance to copy data from. The *pData* parameter specifies where to copy the data, and *nCount* specifies the number of bytes to copy.

Parameters *hInstance* is the instance handle of a previous invocation of the application.

pData is a short pointer to a buffer in the current instance.

nCount is a short integer number specifying the number of bytes to copy.

Return Value *nBytes*, a short integer value, is the number of bytes actually copied.

GetProcAddress (*hModule*, *lpProcName*) : *lpAddress*

Purpose This function retrieves the memory address of the function whose name is pointed to by *lpProcName*. **GetProcAddress** searches for the function in the module specified by *hModule*, or in the current module if *hModule* is NULL. The function must be an exported function; the module's definition file must contain an appropriate **EXPORTS** line for the function.

The retrieved address points to prolog code that is executed before the function is executed. The prolog code loads the data segment of the specified module instance. Thus, when the function is invoked using the retrieved address, the correct data segment will be loaded before the function is executed.

Parameters *hModule* is a module handle or an instance handle of the module containing the function. If *hModule* is a module handle, the most recent instance of the module is used.

lpFunctionName is a long pointer to a character string naming the function. The string must be a null-terminated ASCII string.

Return Value *lpAddress* is a long pointer to the function's entry point if the function is successful. Otherwise, it is NULL.

Note **GetProcAddress** should be used to retrieve addresses of exported functions belonging to other modules only. The **MakeProcInstance** function can be used to access functions within different instances of the current module.

GetCodeHandle (*lpProc*) : *hCode*

Purpose This function returns the handle of the code segment containing the function pointed to by *lpProc*.

Parameters *lpProc* is a long pointer to a function.

Return Value *hCode* is the handle of the code segment containing the function.

Notes If the code segment containing the function is already loaded, **GetCodeHandle** marks the segment as recently used. If the code segment is not loaded, **GetCodeHandle** attempts to load it. Thus, an application can use this function to attempt to preload one or more segments needed to perform a particular task.

MakeProcInstance (*lpProc*, *hInstance*) : *lpAddress*

Purpose This function binds the data segment of the module instance specified by *hInstance* to the function pointed to by *lpProc*.

The address returned by **MakeProcInstance** points to prolog code that is executed before the function is executed. The prolog code loads the data segment of the specified module instance. Thus, when the function is invoked using the retrieved address, the correct data segment will be loaded before the function is executed.

Parameters *lpProc* is the long pointer, before binding, to the desired function.

hInstance is the instance handle associated with the desired data segment.

Return Value *lpAddress* is the long pointer, after binding, to the function if the function is successful. Otherwise, it is NULL.

Notes

MakeProcInstance should be used to access functions from instances of the current module only. The function is not required if the module has a single data segment for all instances. **GetProcAddress** should be used to access functions exported by other modules.

After **MakeProcInstance** has been called for a particular function, all calls to the function should be made through the retrieved address.

FreeProcInstance (*lpProc*)

Purpose

This function frees the function specified by *lpProc* from the data segment bound to it by the **MakeProcInstance** function. When the function is subsequently invoked, it uses its default data segment.

Parameters

lpProc is a long pointer to a function. The address must have been previously created using the **MakeProcInstance** function.

Return Value None.

LoadLibrary (*lpLibFileName*) : *hLibModule*

Purpose

This function loads the library module contained in the specified file and returns a handle to the loaded module.

Parameter

lpLibFileName is a long pointer to a string specifying the name of the library file. The string must be a null-terminated ASCII string.

Return Value

hLibModule is a handle to the loaded library module. It is NULL if the given file is not a library file. If the return value is less than 32, the value represents an MS-DOS system call error code (2, 8, or 11).

FreeLibrary (*hLibModule*)

Purpose

This function decreases the reference count of the loaded library module by one. When the reference count reaches zero, the memory occupied by the module is freed.

Parameter

hLibModule is a handle to a loaded library module.

Return Value None.

GetVersion () : wVersion

Purpose This function returns the current version of Windows.

Parameters None.

Return Value *wVersion* is an unsigned short integer value specifying the major and minor version number of Windows. The high-order byte is the minor version (revision) number; the low-order byte is the major version number.

Catch (*lpCatchBuf*) : *nThrowBack*

Purpose This function catches the current execution environment and copies it to the buffer pointed to by *lpCatchBuf*. The buffer can then be used by the **Throw** function to restore the execution environment to the saved values. The execution environment includes the state of all system registers and instruction counter.

Parameters *lpCatchBuf* is a long pointer to a buffer having CATCHBUF type.

Return Value *nThrowBack*, a short integer value, is zero if the environment is copied to the buffer.

Throw (*lpCatchBuf*, *nThrowBack*)

Purpose This function restores the execution environment to the values saved in the buffer pointed to by *lpCatchBuf*. Execution then continues at the **Catch** function that copied the environment to *lpCatchBuf*. The *nThrowBack* parameter is passed as the return value to the **Catch** function. It can be any nonzero integer value.

Parameters *lpCatchBuf* is a long pointer to a buffer having CATCHBUF type.

nThrowBack is a short integer to be returned to the **Catch** function.

Return Value None.

4.3 Memory Manager Functions

This section describes the functions used to allocate and manage global and local memory. Windows provides two heaps for memory allocation during program execution: the global heap and the application's local heap.

The global heap is all of system memory that is not explicitly allocated for Windows or applications. Any application can allocate one or more blocks of memory from the global heap. Global memory blocks can be any size.

The local heap is free memory in the application's data segment. The user specifies the size of this heap in the application's module definition file.

The application can allocate one or more blocks of local heap, but no block may be larger than available memory. The maximum size for any local heap is 64K bytes.

The application is responsible for freeing any memory it has allocated.

GlobalAlloc (*wFlags*, *dwBytes*) : *hMem*

Purpose This function allocates *dwBytes* of memory from the global heap. The memory can be fixed or moveable, depending on the memory type specified by *Flags*.

Parameters *wFlags* is an unsigned short integer flag value specifying how to allocate the memory. It can be any one of the following:

Value	Meaning
GMEM_FIXED	Memory is fixed (the default).
GMEM_MOVEABLE	Memory is moveable.
GMEM_ZEROINIT	Memory contents are initialized to zero.
GMEM_DISCARDABLE	Memory is discardable.
GMEM_NODISCARD	Discarding of objects will not be performed in order to satisfy the allocation request.
GMEM_NOCOMPACT	Memory compaction and discarding will not be performed in order to satisfy the allocation request.

dwBytes is an unsigned long integer value specifying the number of bytes to allocate.

Return Value *hMem* is a global memory handle if the function is successful. Otherwise, it is NULL.

GlobalCompact (*dwMinFree*) : *dwLargest*

Purpose This function generates *dwMinFree* free bytes of global memory by compacting, if necessary, the system's global heap. The function checks global heap for *dwMinFree* contiguous free bytes. If the bytes do not exist, **GlobalCompact** compacts memory by first moving all unlocked moveable blocks into high memory. If this does not generate the requested amount of space, the function discards unlocked discardable moveable blocks until the requested space is available.

Parameters *dwMinFree* is an unsigned long integer specifying the number of free bytes desired. If *dwMinFree* is zero, the function returns a value but does not compact memory.

Return Value *dwLargest* is an unsigned long integer specifying the number of bytes in the largest block of free global memory.

GlobalDiscard (*hMem*) : *hOldMem*

Purpose This function discards the global memory block specified by *hMem*.

Parameters *hMem* is a handle to a global memory block.

Return Value *hOldMem* is NULL if the function is successful. Otherwise, it is equal to *hMem*.

GlobalFree (*hMem*) : *hOldMem*

Purpose This function frees the global memory block identified by *hMem*.

Parameters *hMem* is a handle to the global memory block to be freed.

Return Value *hOldMem* is NULL if the function is successful. Otherwise, it is equal to *hMem*.

Notes **GlobalFree** must not be used to free a locked memory block. The block must first be unlocked using the **GlobalUnlock** function.

GlobalLock (*hMem*) : *lpAddress*

Purpose This function retrieves the absolute memory address of the global memory block specified by *hMem*. The block is locked into memory at the given address and its reference count is incremented by one. Locked memory is not subject to moving or discarding.

The block remains locked in memory until its reference count is decremented to zero using the **GlobalUnlock** function.

Parameters *hMem* is a handle to the global memory block to be locked.

Return Value *lpAddress* is a long pointer to the first byte of memory in the global block if the function is successful. If the object has been discarded or there is an error, *lpAddress* is NULL.

Notes Discarded objects always have a reference count of zero.

GlobalReAlloc (*hMem*, *dwBytes*, *wFlags*) : *hNewMem*

Purpose This function reallocates the global memory block specified by *hMem* by increasing or decreasing its size to the number of bytes specified by *dwBytes*.

Parameters *hMem* is a handle to the global memory block to be reallocated.

dwBytes is an unsigned long integer specifying the new size of the memory block.

wFlags is an unsigned short integer flag specifying how to reallocate the global block. It can be one or more of the following flags, joined with the bitwise OR operator:

Value	Meaning
GMEM_FIXED	Memory is fixed (the default).
GMEM_MOVEABLE	Memory is moveable. If <i>dwBytes</i> is zero, this flag causes a previously fixed block to be freed or a previously moveable object to be discarded (if the block's reference count is zero). If <i>dwBytes</i> is nonzero and the block specified by <i>hMem</i>

		is fixed, this flag allows the reallocated block to be moved to a new fixed location. (Note that the handle returned by GlobalReAlloc in this case may be different from the handle passed to the function.)
	GMEM_ZEROINIT	If the block is growing, the additional memory contents are initialized to zero.
	GMEM_DISCARDABLE	Memory is discardable.
	GMEM_MODIFY	Memory flags are modified. Can be used only with GMEM_DISCARDABLE. The <i>InBytes</i> parameter is ignored.
	GMEM_NODISCARD	Discarding of objects will not be performed in order to satisfy the allocation request.
	GMEM_NOCOMPACT	Memory compaction and discarding will not be performed in order to satisfy the allocation request.

Return Value *hNewMem* is a handle to the reallocated global memory if the function is successful. *hNewMem* is NULL if the block cannot be reallocated.

hNewMem is always identical to the *hMem* parameter, unless the GMEM_MOVEABLE flag is used to allow movement of a fixed block to a new fixed location.

GlobalSize (*hMem*) : *dwBytes*

Purpose This function retrieves the current size, in bytes, of the global memory block specified by *hMem*.

Parameters *hMem* is a handle to a global memory block.

Return Value *dwBytes* is an unsigned long integer specifying the size, in bytes, of the specified memory block. It is 0 if the given handle is not valid.

Notes The actual size of a memory block is sometimes larger than the size requested when the memory was allocated.

GlobalUnlock (*hMem*) : *bResult*

Purpose This function unlocks the global memory block specified by *hMem* and decrements the block's reference count by 1. The block is completely unlocked, and subject to moving or discarding, if the reference count is decremented to 0.

Parameters *hMem* is a handle to the global memory block to be unlocked.

Return Value *bResult*, a Boolean value, is nonzero if the block is still locked. It is zero if the block is now unlocked or if the *hMem* parameter was invalid.

GlobalFlags (*hMem*) : *wFlags*

Purpose This function returns information about the specified global memory block.

Parameters *hMem* is a handle to a global memory block.

Return Value *wFlags*, an unsigned short integer, contains one of the following memory allocation flags in the high-order byte:

Value	Meaning
GMEM_SWAPPED	The block has been swapped.
GMEM_DISCARDED	The block has been discarded.
GMEM_DISCARDABLE	The block is marked as discardable.

The low-order byte of *wFlags* contains the reference count of the block. The GMEM_LOCKCOUNT mask may be used to retrieve the lock count value from *wFlags*.

LocalAlloc (*wFlags*, *wBytes*) : *hMem*

Purpose This function allocates *wBytes* bytes of memory from the local heap. The memory block can be either fixed or movable as specified by *wFlags*.

Parameters *wFlags* is an unsigned short integer flag specifying how the memory is to be allocated. It can be any one of the following:

Value	Meaning
LMEM_FIXED	Memory is fixed (the default).
LMEM_MOVEABLE	Memory is moveable.
LMEM_ZEROINIT	Memory contents are initialized to zero.
LMEM_DISCARDABLE	Memory is discardable.
LMEM_NODISCARD	Discarding of objects will not be performed in order to satisfy the allocation request.
LMEM_NOCOMPACT	Memory compaction and discarding will not be performed in order to satisfy the allocation request.

wBytes is an unsigned short integer value specifying the total number of bytes to allocate.

Return Value *hMem* is a handle to the newly allocated local memory block if the function is successful. Otherwise, it is NULL.

LocalCompact (*wMinFree*) : *wLargest*

Purpose This function generates *wMinFree* free bytes of memory by compacting, if necessary, the module's local heap. The function checks local heap for *wMinFree* contiguous free bytes. If the bytes do not exist, **LocalCompact** compacts local memory by first moving all unlocked moveable blocks into high memory. If this does not generate the requested amount of space, the function discards discardable moveable blocks that are not locked down until the requested space is available.

Parameters *wMinFree* is an unsigned short integer value specifying the number of free bytes desired. If *wMinFree* is 0, the function returns a value but does not compact memory.

Return Value *wLargest* is an unsigned short integer number specifying the number of bytes in the largest block of free local memory.

LocalDiscard (*hMem*) : *hOldMem*

Purpose This function discards the local memory block specified by *hMem*.

Parameters *hMem* is a handle to a local memory block.

Return Value *hOldMem* is NULL if the function is successful. Otherwise, it is equal to *hMem*.

LocalFree (*hMem*) : *hOldMem*

Purpose This function frees the local memory block identified by *hMem*.

Parameters *hMem* is a handle to the local memory block to be freed.

Return Value *hOldMem* is NULL if the function is successful. Otherwise, it is equal to *hMem*.

LocalLock (*hMem*) : *pAddress*

Purpose This function locks the local memory block specified by *hMem*. The block is locked into memory at the given address and its reference count is incremented by one. Locked memory is not subject to moving or discarding.

The block remains locked in memory until its reference count is decremented to zero using the **LocalUnlock** function.

Parameters *hMem* is a handle to the local memory block to be locked.

Return Value *pAddress* is a short pointer to the first byte of memory in the local block if the function is successful. Otherwise, it is NULL.

LocalFreeze (*Dummy*)

Purpose This function prevents the local heap from being compacted. The **LocalMelt** function can later be called to allow compacting.

Parameters *Dummy* is not used and can be set to zero.

Return Value None.

LocalMelt (*Dummy*)

Purpose This function allows compacting of the local heap. It can be used after **LocalFreeze** has been called.

Parameters *Dummy* is not used and can be set to zero.

Return Value None.

LocalReAlloc (*hMem*, *wBytes*, *wFlags*) : *hNewMem*

Purpose This function reallocates the local memory block specified by *hMem* by increasing or decreasing its size to the number of bytes specified by *wBytes*.

Parameters *hMem* is a handle to the local memory block to be reallocated.

wBytes is an unsigned short integer value specifying the new size of the memory block.

wFlags is an unsigned short integer value specifying how to allocate the local block. It consists of one or more of the following flags, joined with the bitwise OR operator:

Value	Meaning
LMEM_FIXED	Memory is fixed (the default).
LMEM_MOVEABLE	Memory is moveable. If <i>wBytes</i> is zero, this flag causes a previously fixed block to be freed or a previously moveable object to be discarded (if the block's reference count is zero).
	If <i>wBytes</i> is nonzero and the block specified by <i>hMem</i> is fixed, this flag allows the reallocated block to be moved to a new fixed location. (Note that the handle returned by LocalReAlloc in this case may be different from the handle passed to the function.)

LMEM_ZEROINIT	If the block is growing, the additional memory contents are initialized to zero.
LMEM_DISCARDABLE	Memory is discardable.
LMEM_MODIFY	Memory flags are modified. Can be used with LMEM_DISCARDABLE only. The <i>nBytes</i> parameter is ignored.
LMEM_NODISCARD	Discarding of objects will not be performed in order to satisfy the allocation request.
LMEM_NOCOMPACT	Memory compaction and discarding will not be performed in order to satisfy the allocation request.

Return Value *hNewMem* is a handle to the reallocated global memory if the function is successful. *hNewMem* is NULL if the block cannot be reallocated.

hNewMem is always identical to the *hMem* parameter, unless the LMEM_MOVEABLE flag is used to allow movement of a fixed block to a new fixed location.

LocalSize (*hMem*) : *wBytes*

Purpose This function retrieves the current size, in bytes, of the local memory block specified by *hMem*.

Parameters *hMem* is a handle to a local memory block.

Return Value *wBytes* is an unsigned short integer specifying the size, in bytes, of the specified memory block. It is NULL if the given handle is not valid.

Notes The actual size of a memory block is sometimes larger than the size requested when the memory was allocated.

LocalUnlock (*hMem*) : *bResult*

- Purpose* This function unlocks the local memory block specified by *hMem* and decrements the block's reference count by 1. The block is completely unlocked, and subject to moving or discarding, if the reference count is decremented to zero.
- Parameters* *hMem* is a handle to the local memory block to be unlocked.
- Return Value* *bResult*, a Boolean value, is nonzero if the block is still locked. It is zero if the block is now unlocked or if the *hMem* parameter was invalid.

LocalHandleDelta (*nNewDelta*) : *nCurrentDelta*

- Purpose* This function sets the number of handle table entries to be allocated when the local heap manager runs out of handle table entries for local moveable objects.
- Parameter* *nNewDelta*, a short integer value, is the number of handle table entries to be allocated (the "handle delta"). If *nNewDelta* is 0, the current handle delta is returned.
- Return Value* *nCurrentDelta* is a short integer value specifying the current handle delta.

LockData (*Dummy*) : *hMem*

- Purpose* This function locks the current data segment in memory. The function is intended to be used in modules that have moveable data segments.
- Parameters* *Dummy* is not used and can be set to zero.
- Return Value* *hMem* is a handle to the locked data segment if the function is successful. Otherwise, it is NULL.

UnlockData (*Dummy*)

- Purpose* This function unlocks the current data segment. It is intended to be used by modules that have moveable data segments.
- Parameters* *Dummy* is not used and can be set to 0.
- Return Value* None.

LocalFlags (*hMem*) : *wFlags*

Purpose This function returns information about the specified local memory block.

Parameters *hMem* is a handle to a local memory block.

Return Value *wFlags*, an unsigned short integer, contains one of the following memory allocation flags in the high-order byte:

Value	Meaning
LMEM_DISCARDED	The block has been discarded.
LMEM_DISCARDABLE	The block is marked as discardable.

The low-order byte of *wFlags* contains the reference count of the block. The LMEM_LOCKCOUNT mask may be used to retrieve the lock count value from *wFlags*.

4.4 Task Functions

This section describes the functions used to manage tasks. A task is any program that executes as an independent unit. All applications are executed as tasks. Each instance of an application is a task.

GetCurrentTask () : *hTask*

Purpose This function returns the handle of the currently executing task.

Parameters None.

Return Value *hTask* is a task handle if the function is successful. Otherwise, it is NULL.

Yield () : *bResult*

Purpose This function halts the current task and starts any waiting task.

Parameters None.

Return Value *bResult*, a Boolean value, is nonzero if a waiting task is started. Otherwise, it is zero.

Note Applications that contain windows should use a **PeekMessage**, **TranslateMessage**, **DispatchMessage** loop rather than calling **Yield** directly. The **PeekMessage** loop handles message synchronization properly and yields at the appropriate times.

SetPriority (*hTask*, *nChangeAmount*) : *nNew*

Purpose This function modifies the task priority of the task specified by *hTask* by adding *nChangeAmount* to the current priority. The task priority can be modified to any value in the range -15 to 15. The highest task priority is -15, the lowest is 15. Task priority is initially set to 0.

Parameters *hTask* is a handle to the task to be set.

nChangeAmount is a short integer value specifying the amount to change the priority.

Return Value *nNew* is a short integer value specifying the new priority of the task.

4.5 Resource Manager Functions

This section describes the functions used to find and load program resources from a Windows executable file. The functions let applications load cursors, icons, bitmaps, strings, fonts, and other useful resources into memory.

AddFontResource (*lpFilename*) : *nFonts*

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

Parameters *lpFilename* is either a long pointer to a string naming the font resource file, or a handle to a loaded module in the low-order word and zero in the high-order word. If *lpFilename* is a long pointer to the font resource file, the string must be null-terminated, have the DOS file name format, and include the extension.

Return Value *nFonts* is a short integer specifying the number of fonts added. *nFonts* is zero if no fonts are loaded.

Notes Any application that adds or removes fonts from the Windows font table should notify other windows of the change by sending a WM_FONTCHANGE message (using **SendMessage**). The message should be sent to all window handles enumerated by the **EnumWindows** function.

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

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

RemoveFontResource (*lpFilename*) : *bSuccess*

Purpose This function removes an added font resource from the file named by *lpFilename* from the Windows font table.

Parameters *lpFilename* is either a long pointer to a string naming the font resource file, or a handle to a loaded module in the low-order word and zero in the high-order word. If *lpFilename* is a long pointer to the font resource file, the string must be null-terminated, have the DOS file name format, and include the extension.

Return Value *bSuccess*, a Boolean value, is nonzero if the function is successful. Otherwise, it is zero.

Notes Any application that adds or removes fonts from the Windows font table should notify other windows of the change by sending a WM_FONTCHANGE message (using **SendMessage**). The message should be sent to all window handles enumerated by the **EnumWindows** function.

RemoveFontResource may not actually remove the font resource. If there are outstanding references to the resource, the font resource remains loaded until the last referencing logical font has been deleted using the **DeleteObject** function.

LoadBitmap (*hInstance*, *lpBitmapName*) : *hBitmap*

- Purpose* This function loads the bitmap resource named by *lpBitmapName* from the executable file associated with the module *hInstance*. The function loads the resource into memory only if it has not been previously loaded. Otherwise, it retrieves a handle to the existing resource.
- Parameters* *hInstance* is the instance handle of the module whose executable file contains the bitmap.
lpBitmapName is a long pointer to a character string naming the bitmap. The string must be a null-terminated ASCII string.
- Return Value* *hBitmap* is a handle to the specified bitmap. It is NULL if no such bitmap exists.
- Note* **LoadBitmap** assumes that the bitmap is stored in a device-independent format. The function stretches or compresses the bitmap to accommodate the system's aspect ratio and resolution.
The bitmap handle returned by **LoadBitmap** is automatically freed by Windows when the application terminates.

LoadCursor (*hInstance*, *lpCursorName*) : *hCursor*

- Purpose* This function loads the cursor resource named by *lpCursorName* from the executable file associated with the module specified by *hInstance*. The function loads the cursor into memory only if it has not been previously loaded. Otherwise, it retrieves a handle to the existing resource.
- LoadCursor** can also be used to access the predefined cursors used by Windows. The *hInstance* argument must be set to NULL, and *lpCursorName* must be one of the following:

Name	Cursor Type
IDC_ARROW	Standard arrow cursor
IDC_IBEAM	Text I-beam cursor
IDC_WAIT	Hourglass cursor
IDC_UPARROW	Vertical arrow cursor
IDC_CROSS	Crosshair cursor
IDC_SIZE	Size box cursor (used when the user is sizing a window)
IDC_ICON	Icon cursor (black box used for cursor in icon area)

Parameters *hInstance* is the instance handle of the module whose executable file contains the cursor.

lpCursorName is a long pointer to a character string naming the cursor resource. The string must be a null-terminated ASCII string.

Return Value *hCursor* is a handle to the newly loaded cursor if the function is successful. Otherwise, it is NULL.

LoadIcon (*hInstance*, *lpIconName*) : *hIcon*

Purpose This function loads the icon resource named by *lpIconName* from the executable file associated with the module specified by *hInstance*. The function loads the icon only if it has not been previously loaded. Otherwise, it retrieves a handle to the loaded resource.

LoadIcon can also be used to access the predefined icons used by Windows. The *hInstance* argument must be set to NULL, and *lpIconName* must be one of the following:

Name	Icon Type
IDI_APPLICATION	Default application icon
IDI_HAND	Hand-shaped icon (used in severe warning messages)
IDI_QUESTION	Question mark (used in prompting messages)
IDI_EXCLAMATION	Exclamation mark (used in warning messages)
IDI_ASTERISK	Asterisk (used in informative messages)

Parameters *hInstance* is the instance handle of the module whose executable file contains the icon.

lpIconName is a long pointer to a character string naming the icon resource. The string must be a null-terminated ASCII string.

Return Value *hIcon* is a handle to an icon resource if the function is successful. Otherwise, it is NULL.

LoadMenu (*hInstance*, *lpMenuName*) : *hMenu*

Purpose This function loads the menu resource named by *lpMenuName* from the executable file associated with the module specified by *hInstance*.

The function loads the menu only if it has not been previously loaded. Otherwise, it retrieves a handle to the loaded resource.

Parameters *hInstance* is the instance handle of the module whose executable file contains the menu.

lpMenuName is a long pointer to a character string naming the menu resource. The string must be a null-terminated ASCII string.

Return Value *hMenu* is a handle to a menu resource if the function is successful. Otherwise, it is NULL.

LoadString (*hInstance*, *wID*, *lpBuffer*, *nBufferMax*) : *nSize*

Purpose This function loads a string resource identified by *wID* from the executable file associated with module *hInstance*. The function copies the string into the buffer pointed to by *lpBuffer*, and appends a terminating null character.

Parameters *hInstance* is the instance handle of the module whose executable file contains the string resource.

wID is an unsigned short integer value identifying the string to load.

lpBuffer is a long pointer to the buffer to receive the string.

nBufferMax is a short integer value specifying the maximum number of characters to be copied to the buffer. The string is truncated if it is longer than the buffer.

Return Value *nSize* is a short integer value specifying the actual number of characters copied into the buffer. It is 0 if the string resource does not exist.

LoadAccelerators (*hInstance*, *lpTableName*) : *hRes*

Purpose This function loads the accelerator table named by *lpTableName* from the executable file associated with the module specified by *hInstance*.

The function loads the table only if it has not been previously loaded. Otherwise, it retrieves a handle to the loaded table.

Parameters *hInstance* is the instance handle of the module whose executable file contains the accelerator table.

lpTableName is a long pointer to a string specifying the name of the accelerator table. The string must be a null-terminated ASCII string.

Return Value *hRes* is a handle to the loaded accelerator table if the function is successful. Otherwise, it is NULL.

FindResource (*hInstance*, *lpName*, *lpType*) : *hResInfo*

Purpose This function determines the location of a resource in the specified resource file. The *lpName* and *lpType* parameters define the name and resource type.

Parameters *hInstance* is the instance handle of the module whose executable file contains the resource.

lpName is a long pointer to a null-terminated string representing the name of the resource.

lpType is a long pointer to a null-terminated string representing the type name of the resource. For predefined resource types, the *lpType* parameter should be one of the following values:

Type	Meaning
RT_CURSOR	Cursor resource
RT_BITMAP	Bitmap resource
RT_ICON	Icon resource
RT_MENU	Menu resource
RT_DIALOG	Dialog box
RT_STRING	String resource
RT_FONT	Font resource
RT_ACCELERATOR	Accelerator table

Return Value *hResInfo* is a handle to the named resource. It is NULL if the requested resource cannot be found.

Notes If the high-order word of *lpName* or *lpType* is 0, the low-order word specifies the integer ID of the name or type of the given resource. Otherwise, the parameters are long pointers to null-terminated strings. If the first character of the string is a pound sign (#), the remaining characters represent a decimal number that specifies the integer ID of the resource's name or type. For example, the string "#258" represents the integer ID 258.

To reduce the amount of memory required for the resources used by an application, the application should refer to the resources by integer ID instead of by name.

LoadResource (*hInstance*, *hResInfo*) : *hResData*

- Purpose* This function loads a resource identified by *hResInfo* from the executable file associated with the module *hInstance*. The function loads the resource into memory only if it has not been previously loaded. Otherwise, it retrieves a handle to the existing resource.
- Parameters* *hInstance* is the instance handle of the module whose executable file contains the resource.
hResInfo is the resource handle associated with the desired resource. This handle is assumed to have been created using the **FindResource** function.
- Return Value* *hResData* is a handle to the data associated with the resource. It is NULL if no such resource exists.
- Note* The resource is not actually loaded until **LockResource** is called to translate the handle returned by **LoadResource** into a far pointer to the resource data.

AllocResource (*hInstance*, *hResInfo*, *nSize*) : *hMem*

- Purpose* This function allocates uninitialized memory for the passed resource. All resources must be initially allocated with this function. The **LoadResource** function calls this function before loading the resource.
- Parameters* *hInstance* is the instance handle of the module whose executable file contains the resource.
hResInfo is the resource handle associated with the desired resource. This handle is assumed to have been created using the **FindResource** function.
nSize is a short integer value specifying an override size. It is ignored if zero.
- Return Value* *hMem* is a handle to the allocated memory.

LockResource (*hResInfo*) : *lpResInfo*

- Purpose* This function retrieves the absolute memory address of the loaded resource identified by *hResInfo*. The resource is locked in memory and the given address and its reference count is increased by one. The locked resource is not subject to moving or discarding.

The resource remains locked in memory until its reference count is decreased to zero through calls to the **FreeResource** function.

If the resource identified by *hResInfo* has been discarded, the resource handler function (if any) associated with the resource is called before **LockResource** returns. The resource handler function can recalculate and reload the resource if desired. After the resource handler function returns, **LockResource** makes another attempt to lock the resource and returns with the result.

Parameters *hResInfo* is the resource handle associated with the desired resource. This handle is assumed to have been created using the **FindResource** function.

Return Value *lpResInfo* is a long pointer to the first byte of the loaded resource if the resource was locked. Otherwise, *lpResInfo* is NULL.

FreeResource (*hResData*) : *bFreed*

Purpose This function removes a loaded resource from memory by freeing the allocated memory occupied by that resource.

FreeResource does not actually free the resource until the reference count is zero (that is, the number of calls to the function equals the number of times the application called **LoadResource** for this resource). This ensures that the data remains in memory while the application is still using it.

Parameters *hResData* is a handle to the data associated with the resource. The handle is assumed to have been created using the **LoadResource** function.

Return Value *bFreed*, a Boolean value, is nonzero if the function has failed and the resource has not been freed. It is zero if the function is successful.

AccessResource (*hInstance*, *hResInfo*) : *hFile*

Purpose This function opens the specified resource file and moves the file pointer to the beginning of the specified resource, letting an application read the resource from the file into its own local memory space. The function supplies a DOS file handle that can be used in subsequent file read calls to load the resource. The file is opened for reading only.

Applications that use this function must close the resource file after reading the resource.

Parameters *hInstance* is the module instance handle of the module whose executable file contains the resource.

hResInfo is the resource handle associated with the desired resource. This handle is assumed to have been created using the **FindResource** function.

Return Value *hFile* is a DOS file handle to the specified resource file. It is -1 if the resource cannot be found.

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

SizeofResource (*hInstance*, *hResInfo*) : *nBytes*

Purpose This function supplies the size in bytes of the specified resource. It is typically used with the **AccessResource** to prepare local memory to receive a resource from the file.

Parameters *hInstance* is the instance handle of the module whose executable file contains the resource.

hResInfo is the resource handle associated with the desired resource. This handle is assumed to have been created using the **FindResource** function.

Return Value *nBytes* is a short integer value specifying the number of bytes in the resource. It is zero if the resource cannot be found.

SetResourceHandler (*hInstance*, *lpType*, *lpLoadFunc*) : *lpLoadFunc*

Purpose This function sets up a function to load resources. It is used internally by Windows to implement calculated resources. Applications may find this function useful for handling their own resource types, but its use is not required.

The **LockResource** function calls the *lpLoadFunc* whenever it fails to lock a resource having the type given by *lpType*. The *lpLoadFunc* receives information about the resource to be locked and can process that information as desired. After the *lpLoadFunc* returns, **LockResource** attempts the lock the resource once more.

Parameters *hInstance* is the instance handle of the module whose executable file contains the resource.

lpType is a long pointer to a short integer specifying a resource type.

lpLoadFunc is a long pointer to the user-supplied function.

Return Value *lpLoadFunc* is the long pointer to the user-supplied function.

Note The user-supplied function must have the form:

lpLoadFunc(hMem, hInstance, hResInfo)

where *hMem* is a handle to a stored resource, *hInstance* is the instance handle of the module whose executable file contains the resource, and *hResInfo* is the resource handle associated with the desired resource (assumed to have been created using **FindResource**).

hMem is NULL if the resource has not yet been loaded. If an attempt to lock a block specified by *hMem* fails, the resource has been discarded and must be reloaded.

4.6 String Translation Functions

This section describes functions used to translate strings from one character set to another, to convert the case of strings, and to find adjacent characters in a string.

AnsiToOem (*lpAnsiStr, lpOemStr*) : *bTranslated*

Purpose This function translates the string pointed to by *lpAnsiStr* from the ANSI character set to the OEM-defined character set.

Parameters *lpAnsiStr* is a long pointer to a null-terminated string of characters from the ANSI character set.

lpOemStr is a long pointer to the location to which the translated string is copied. *lpOemStr* can be the same as *lpAnsiStr* to translate the string in place.

Return Value *bTranslated*, a Boolean value, is nonzero if all characters in the ANSI string have been translated into equivalent characters in the OEM character set. It is zero if one or more

characters in the ANSI string had no direct equivalent in the OEM character set. In such cases, an arbitrary OEM character is selected for the translation.

OemToAnsi (*lpOemStr*, *lpAnsiStr*) : *bTranslated*

Purpose This function translates the string pointed to by *lpOemStr* from the OEM-defined character set to the ANSI character set.

Parameters *lpOemStr* is a long pointer to a null-terminated string of characters from the OEM-defined character set.

lpAnsiStr is a long pointer to the location to which the translated string is copied. *lpAnsiStr* can be the same as *lpOemStr* to translate the string in place.

Return Value *bTranslated*, a Boolean value, is nonzero if all characters in the OEM string have been translated into equivalent characters in the ANSI character set. It is zero if one or more characters in the OEM string had no direct equivalent in the ANSI character set. In such cases, an arbitrary ANSI character is selected for the translation.

AnsiUpper (*lpStr*) : *cChar*

Purpose This function converts a string or a character to upper case.

Parameter *lpStr* is a long pointer to a null-terminated string, or it is a long integer value whose high-order word contains zero and low-order word a single character.

Return Value *cChar*, a byte value, is the converted character if a single character is given. Otherwise, *cChar* is a long pointer and equal to *lpStr*.

AnsiLower (*lpStr*) : *cChar*

Purpose This function converts the given string to lower case.

Parameter *lpStr* is a long pointer to a null-terminated string, or it is a long integer value whose high-order word contains zero and whose low-order word contains a single character.

Return Value *cChar*, a byte value, is the converted character if a single character is given. Otherwise, *cChar* is a long pointer and equal to *lpStr*.

AnsiNext (*lpCurrentChar*) : *lpNextChar*

Purpose This function moves to the next character in a string.

Parameter *lpCurrentChar* is a long pointer to a character in a null-terminated string.

Return Value *lpNextChar* is a long pointer to the next character in the string, or if there is no next character, to the null character at the end of the string.

Note **AnsiNext** is used to move through strings whose characters are two or more bytes each (for example, strings containing characters from a Japanese character set).

AnsiPrev (*lpStart*, *lpCurrentChar*) : *lpPrevChar*

Purpose This function moves to the previous character in a string.

Parameters *lpStart* is a long pointer to the beginning of the string.

lpCurrentChar is a long pointer to a character in a null-terminated string.

Return Value *lpPrevChar* is a long pointer to the previous character in the string, or to the first character in the string if *lpCurrentChar* is equal to *lpStart*.

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

4.7 Atom Manager Functions

This section describes the functions used to create and use atoms. An atom is an integer number that uniquely identifies a character string stored in the atom table. Atoms are useful in applications that use many character strings.

InitAtomTable (*nSize*) : *bResult*

Purpose This function initializes an atom hash table and sets its size to *nSize*. If this function is not called, the atom hash table size is set to 37 by default, limiting the maximum number of atoms to 37.

If used, this function should be called before any other atom manager function.

Parameter *nSize* is a short integer value specifying the size (in table entries) of the atom hash table.

Return Value *bResult*, a Boolean value, is nonzero if the function is successful. Otherwise, it is zero.

AddAtom (*lpString*) : *nAtom*

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

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

Parameter *lpString* is a long pointer to the character string to be added to the table. The string must be an null-terminated ASCII string.

Return Value *nAtom*, a short integer value, is the newly created atom if the function is successful. Otherwise, it is NULL.

Notes The atom values returned by **AddAtom** are in the range C000 to FFFF (hexadecimal).

DeleteAtom (*nAtom*) : *nOldAtom*

Purpose This function deletes an atom and, if the atom's reference count is zero, removes the associated string from the atom table.

An atom's reference count specifies the number of times the atom has been added to the atom table. **AddAtom** increments the count on each call; **DeleteAtom** decrements the count on each call. **DeleteAtom** removes the string only if the atom's reference count is zero.

Parameter *nAtom* is a short integer value specifying the atom and character string to be deleted.

Return Value *nOldAtom*, a short integer value, is NULL if the function is successful. It is equal to *nAtom* if the function failed and the atom has not been deleted.

FindAtom (*lpString*) : *nAtom*

- Purpose* This function searches the atom table for the character string pointed to by *lpString* and retrieves the atom associated with that string.
- Parameter* *lpString* is a long pointer to the character string to be searched for. The string must be a null-terminated ASCII string.
- Result* *nAtom*, a short integer value, is the atom associated with the given string. It is NULL if the string is not in the table.

GetAtomName (*nAtom*, *lpBuffer*, *nSize*) : *nLength*

- Purpose* This function retrieves a copy of the character string associated with *nAtom* and places it in the buffer pointed to by *lpBuffer*. The *nSize* specifies the maximum size of the buffer.
- Parameters* *nAtom* is a short integer value identifying the character string to be retrieved.
lpBuffer is a long pointer to the buffer to receive the character string.
nSize is a short integer value specifying the maximum size in bytes of the buffer.
- Return Value* *nLength* is a short integer value specifying the actual number of bytes copied to the buffer. It is 0 if the specified atom is not valid.

MAKEINTATOM (*wInteger*) : *nAtom*

- Purpose* This macro creates an integer atom that represents a character string of the form
- $$\# \text{ } digits$$
- where *digits* are the decimal digits of the integer. Integer atoms created by this macro can be added to the atom table using the *AddAtom* function.
- Parameters* *wInteger* is an unsigned short integer value specifying the numeric value of the atom's character string.
- Return Value* *nAtom*, a short integer value, is the atom created for the given integer.

Notes

The **DeleteAtom** function always succeeds for integer atoms, even though it does nothing, and the **GetAtomName** function always returns the string form of the integer atom (e.g. "#123").

4.8 Windows Initialization File Functions

This section describes the functions used to read from and write to the Windows initialization file. The Windows initialization file is a special ASCII file that contains "keyname-value" pairs that represent runtime options for applications.

GetProfileInt (*lpApplicationName*, *lpKeyName*, *nDefault*) : *nKeyValue*

Purpose

This function retrieves the value of an integer key from the Windows initialization file WIN.INI. The function searches the Windows initialization file for a key matching the name specified by *lpKeyName* under the application heading specified by *lpApplicationName*.

An integer entry in the Windows initialization file must have the following form:

[*Application-Name*]
 Key-Name = *value*

where *value* is the key's integer value.

Parameters

lpApplicationName is a long pointer to a character string naming the application. The string must be a null-terminated ASCII string.

lpKeyName is a long pointer to a character string naming a key. The string must be a null-terminated ASCII string.

nDefault is a short integer value specifying the default value for the given key if the key cannot be found in the Windows initialization file.

Result

nKeyValue is a short integer value specifying the value of the given key if the key exists. Otherwise, *nKeyValue* is equal to *nDefault*.

GetProfileString (*lpApplicationName*, *lpKeyName*, *lpDefault*, *lpReturnedString*, *nSize*) : *nLength*

- Purpose* This function copies a character string from the user profile, WIN.INI, into the buffer pointed to by *lpReturnedString*. The function searches the Windows initialization file for a key matching the name specified by *lpKeyName* under the application heading specified by *lpApplicationName*. If the key is found, the corresponding string is copied to the buffer. If the key does not exist, the default character string, *lpDefault*, is copied.
- A string entry in the Windows initialization file must have the following form:

[*Application-Name*]
 Key-Name = *string*

where *string* is an ASCII string.

If *lpKeyName* is NULL, **GetProfileString** enumerates all keynames associated with *lpApplicationName* by filling the location pointed to by *lpReturnedString* with a list of keynames (not values). Each keyname in the list is terminated with a null character. The last string in the list is terminated with two null characters. **GetProfileString** returns the length of the list, up to, but not including, the final null.

- Parameters*
- lpApplicationName* is a long pointer to a character string naming the application. The string must be a null-terminated ASCII string.
- lpKeyName* is a long pointer to a character string naming a key. The string must be a null-terminated ASCII string.
- lpDefault* is a long pointer to the character string to be used if the given key does not exist. It must be a null-terminated ASCII string.
- lpReturnedString* is a long pointer to the buffer to receive the character string.
- nSize* is a short integer value specifying the maximum number of bytes to be copied to the buffer. If the actual string is longer, it is truncated.

Return Value *nLength* is a short integer value specifying the actual number of characters copied to *lpReturnedString*.

Notes **GetProfileString** is not case-dependent, so the strings in *lpApplicationName* and *lpKeyName* may be in any combination of uppercase and lowercase letters.

WriteProfileString (*lpApplicationName*, *lpKeyName*, *lpString*) : *bResult*

Purpose This function copies the character string pointed to by *lpString* into the Windows initialization file, WIN.INI. This function searches the Windows initialization file for the key named by *lpKeyName* under the application heading specified by *lpApplicationName*. If there is no match, it adds a new string entry to the user profile. If there is a matching key, the function replaces that key's value with *lpString*.

If there is no application field for *lpApplicationName*, this function creates a new application field and places an appropriate key-value line in that field of the Windows initialization file.

A string entry in the Windows initialization file has the following form:

[*Application-Name*]
 Key-Name = *string*

where *string* is an ASCII string.

Parameters *lpApplicationName* is a long pointer to a character string naming the application. The string must be a null-terminated ASCII string.

lpKeyName is a long pointer to a character string naming the desired key. The string must be a null-terminated ASCII string.

lpValue is a long pointer to the string to be copied to the file. The string must be a null-terminated ASCII string.

Return Value *bResult*, a Boolean value, is nonzero if the function is successful. Otherwise, it is zero.

Notes Applications that make changes to the WIN.INI file in sections that are also accessed by other applications must send a WM_WININICHANGE message to all applications in the system.

4.9 Debugging Function

This section describes the function used to provide system information on program errors.

FatalExit (*Code*) : *Result*

- Purpose* This function displays the current state of Windows and prompts for instructions on how to proceed. The display includes an error code, *Code*, followed by a symbolic stack trace, showing the flow of execution up to the point of call. The function prompts for user response to an “Abort, Break or Ignore” message. **FatalExit** processes the response as follows:

- a – terminate Windows
- b – simulate an NMI interrupt to enter the debugger
- i – return to the caller

The function is intended to be called whenever Windows detects a fatal error. All input and output is through the computer’s auxiliary port (AUX).

- Parameters* *Code* is an integer value specifying the error code to be displayed.

- Return Value* *Result*, an integer value, is returned only if the user chose to ignore the fatal error.

4.10 Communication Functions

This section describes the functions used to carry out communications through the system’s serial and parallel I/O ports.

OpenComm (*lpComName*, *nInQueue*, *nOutQueue*) : *nCid*

- Purpose* This function opens a communication device and assigns a *nCid* handle to it. The communication device is initialized to a default configuration. The **SetCommState** function should be used to initialize the device to alternate values. The function allocates space for “receive” and “transmit” queues. The queues are used by the interrupt-driven transmit/receive software.

Parameters *lpComName* is a long pointer to a string that contains "COMn" or "LPTn", where *n* is allowed to range from 1 to the number of communication devices available for the type of I/O port.

nInQueue is a short integer value specifying the size of the receive queue.

nOutQueue is a short integer value specifying the size of the transmit queue.

Return Value *nCid* is a short integer value identifying the opened communication device. If an error occurs, *nCid* is one of the following negative error values:

Value	Error
IE_BADID	Invalid or unsupported ID
IE_OPEN	Device already open
IE_NOOPEN	Device not open
IE_MEMORY	Unable to allocate queues
IE_DEFAULT	Error in default parameters
IE_HARDWARE	Hardware not present
IE_BYTESIZE	Invalid byte size
IE_BAUDRATE	Unsupported baud rate

CloseComm (*nCid*) : *nResult*

Purpose This function closes the communication device specified by *nCid* and frees any memory allocated for the device's transmit and receive queues. All characters in the output queue are sent before the communications device is closed.

Parameters *nCid* is a short integer value specifying the device to be closed.

Return Value *nResult* is a short integer value specifying the result of the function. It is 0 if the device is closed. It is negative if an error occurred.

ReadComm (*nCid*, *lpBuf*, *nSize*) : *nBytes*

- Purpose* This function attempts to read *nSize* characters from the communication device specified by *nCid* and copy the characters into the buffer pointed to by *lpBuf*.
- Parameters* *nCid* is a short integer value specifying the communication device to be read.
lpBuf is a long pointer to the buffer to receive the characters read.
nSize is a short integer value specifying the number of characters to read.
- Return Value* *nBytes* is a short integer value specifying the number of characters actually read. It is less than *nSize* only if the number of characters in the receive queue is less than *nSize*. If it is equal to *nSize*, additional characters may be queued for the device. If *nBytes* is 0, no characters are present.
On an error, *nBytes* is set to a value less than 0, making the absolute value of *nBytes* the actual number of characters read. The cause of the error can be determined by using the **GetCommError** function to retrieve the error code and status. Since errors can occur when no bytes are present, if *nBytes* is 0, the **GetCommError** function should be used to ensure that no error occurred.
For parallel I/O ports, *nBytes* will always be 0.

WriteComm (*nCid*, *lpBuf*, *nSize*) : *nBytes*

- Purpose* This function writes *nSize* bytes to the communication device specified by *nCid* from the buffer pointed to by *lpBuf*.
- Parameters* *nCid* is a short integer value specifying the device to receive the characters.
lpBuf is a long pointer to the buffer containing the characters to be written.
nSize is a short integer value specifying the number of characters to write.
- Return Value* *nBytes* is a short integer value specifying the number of characters actually written. On an error, *nBytes* is set to a value less than 0, making the absolute value of *nBytes* the actual number of characters written. The cause of the error can be determined by using the **GetCommError** function to retrieve the error code and status.

UngetCommChar (*nCid*, *cChar*) : *nResult*

Purpose This function places the character *cChar* back into the receive queue, making this character the first to be read on a subsequent read from the queue.

Consecutive calls to **UngetCommChar** are not allowed. The character placed back into the queue must be read before attempting to place another.

Parameters *nCid* is a short integer value specifying the communication device to receive the character.

cChar is the byte value of the character to be put into the receive queue.

Return Value *nResult*, a short integer value, is zero if the function was successful. If an error occurred, *nResult* is negative.

TransmitCommChar (*nCid*, *cChar*) : *nResult*

Purpose This function marks the character *cChar* for immediate transmission by placing it at the head of the transmit queue.

Parameters *nCid* is a short integer value specifying the communication device to receive the character.

cChar is the byte value of the character to be written.

Return Value *nResult* is a short integer value specifying the result of the function. It is 0 if the function is successful. It is negative if the character cannot be transmitted. A character cannot be transmitted if the character specified by the previous **TransmitCommChar** has not yet been sent.

BuildCommDCB (*lpDef*, *lpDCB*) : *nResult*

Purpose This function translates the definition string specified by *lpDef* into appropriate device control block codes and places these codes into the block pointed to by *lpDCB*.

Parameters *lpDef* is a long pointer to a null-terminated string specifying the device control information for a device. The string must have the same form as the MS-DOS **MODE** command line argument.

lpDCB is a long pointer to a data structure having **DCB** type.

Return Value *nResult* is a short integer value specifying the result of the function. It is 0 if the string is translated. It is negative if an error occurs.

GetCommState (*nCid*, *lpDCB*) : *nResult*

Purpose This function fills the buffer pointed to by *lpDCB* with the device control block of the communication device specified by *nCid*.

Parameters *nCid* is a short integer value specifying the device to be examined.

lpDCB is a long pointer to a data structure having **DCB** type.

Return Value *nResult*, a short integer value, is zero if the function was successful. If an error occurred, *nResult* is negative.

SetCommState (*lpDCB*) : *nResult*

Purpose This function sets a communication device to the state specified by the device control block pointed to by *lpDCB*. The device to be set must be identified by the *ID* field of the control block.

This function reinitializes all hardware and control as defined by *lpDCB*, but does not empty transmit or receive queues.

Parameters *lpDCB* is a long pointer to a data structure having **DCB** type.

Return Value *nResult*, a short integer value, is zero if the function is successful. If an error occurred, *nResult* is negative.

GetCommError (*nCid*, *lpStat*) : *nError*

Purpose This function fills the status buffer pointed to by *lpStat* with the current status of the communication device specified by *nCid*. It also returns the error codes that have occurred since the last **GetCommError** call (see Table 4.1 for error codes). If *lpStat* is NULL, only the error code is returned.

Parameters *nCid* is a short integer value specifying the communication device to be examined.

lpStat is a long pointer to a **COMSTAT** data structure to receive the device status.

Return Value *nError* is a short integer value specifying the error codes returned by the most recent communications function.

Table 4.1
Communications Driver Error Codes

Code	Meaning
CE_MODE	Requested mode is not supported, or the given <i>nCid</i> is invalid. If set, this is the only valid error.
CE_RXOVERRUN	Receive queue overflow. There was either no room in the input queue or a character was received after the <i>EofChar</i> was received.
CE_OVERRUN	A character was not read from the hardware before the next character arrived. The character was lost.
CE_RXPARITY	The hardware detected a parity error.
CE_FRAME	The hardware detected a framing error.
CE_BREAK	The hardware detected a break condition.
CE_CTS TO	Clear To Send timeout. CTS was low for the duration specified by <i>CtsTimeout</i> while trying to transmit a character.
CE_DSRTO	Data Set Ready timeout. DSR was low for the duration specified by <i>DsrTimeout</i> while trying to transmit a character.
CE_RLSDTO	Receive Line Signal Detect timeout. RLSD was low for the duration specified by <i>RlsdTimeout</i> while trying to transmit a character.
CE_TXFULL	The transmit queue was full while trying to queue a character.
CE_PTO	Timeout occurred while trying to communicate with a parallel device.
CE_IOE	An I/O error occurred while trying to communicate with a parallel device.
CE_DNS	The parallel device was not selected.
CE_OOP	The parallel device signaled it was out of paper.

SetCommEventMask (*nCid*, *nEvtMask*) : *lpEvent*

Purpose This function enables and retrieves the event mask of the communication device specified by *nCid*. The bits of the *nEvtMask* parameter define which events are to be enabled. The return value points to the current state of the event mask.

Parameters *nCid* is a short integer value specifying the communication device to be examined.

nEvtMask is a short integer value whose bits specify which events are to be enabled. It can be any combination of the following:

Value	Meaning
EV_RXCHAR	Set when any character is received and placed in the receive queue.
EV_RXFLAG	Set when the event character is received and placed in the receive queue. The event character is specified in the device's control block.
EV_TXEMPTY	Set when the last character in the transmit queue is sent.
EV_CTS	Set when the Clear To Send (CTS) signal changes state.
EV_DSR	Set when the Data Set Ready (DSR) signal changes state.
EV_RLSD	Set when the Receive Line Signal Detect (RLSD) signal changes state.
EV_BREAK	Set when a break is detected on input.
EV_ERR	Set when a line status error occurs. Line status errors are CE_RXPARITY, CE_OVERRUN, and CE_FRAME.
EV_RING	Set when ring indicator is detected.
EV_PERR	Set when a printer error is detected on a parallel device. Printer errors are CE_PTO, CE_IOE, CE_DNS, and CE_LOOP.

Return Value *lpEvent* is a long pointer to the event mask, a short integer. Each bit in the event mask specifies whether or not a given event has occurred. A bit is 1 if the event has occurred.

GetCommEventMask (*nCid*, *nEvtMask*) : *nEvent*

Purpose This function retrieves the value of the current event mask, then clears the mask. This function must be used to prevent loss of an event.

Parameters *nCid* is a short integer value specifying the communication device to be examined.

nEvtMask is a short integer value whose bits specify which events are to be enabled. It can be any combination of the following:

Value	Meaning
EV_RXCHAR	Set when any character is received and placed in the receive queue.
EV_RXFLAG	Set when the event character is received and placed in the receive queue. The event character is specified in the device's control block.
EV_TXEMPTY	Set when the last character in the transmit queue is sent.
EV_CTS	Set when the Clear To Send (CTS) signal changes state.
EV_DSR	Set when the Data Set Ready (DSR) signal changes state.
EV_RLSD	Set when the Receive Line Signal Detect (RLSD) signal changes state.
EV_BREAK	Set when a break is detected on input.
EV_ERR	Set when a line status error occurs.

Return Value *nEvent* is a short integer value specifying the current event mask value. Each bit in the event mask specifies whether or not a given event has occurred. A bit is 1 if the event has occurred.

FlushComm (*nCid*, *nQueue*) : *nResult*

Purpose This function flushes all characters from the transmit or receive queue of the communication device specified by *nCid*. The *nQueue* parameter specifies which queue is to be flushed.

Parameters *nCid* is a short integer value specifying the communication device to be flushed.

nQueue is a short integer value specifying which queue is to be flushed. It can be one of the following:

- 0 - Flush transmit queue
- 1 - Flush receive queue

Return Value *nResult* is a short integer value specifying the result of the function. It is 0 if it is successful. It is negative if *nCid* is not a valid device, or if *nQueue* is not a valid queue.

SetCommBreak (*nCid*) : *nResult*

Purpose This function suspends character transmission and places the transmission line in a break state until the **ClearCommBreak** function is called.

Parameters *nCid* is a short integer value specifying the communication device to be suspended.

Return Value *nResult* is a short integer value specifying the result of the function. It is 0 if it is successful. It is negative if *nCid* is not a valid device.

ClearCommBreak (*nCid*) : *nResult*

Purpose This function restores character transmission and places the transmission line in a nonbreak state.

Parameters *nCid* is a short integer value specifying the communication device to be restored.

Return Value *nResult* is a short integer value specifying the result of the function. It is 0 if it is successful. It is negative if *nCid* is not a valid device.

EscapeCommFunction (*nCid*, *nFunc*) : *nResult*

Purpose This function directs the communication device specified by *nCid* to carry out the extended function specified by *nFunc*.

Parameters *nCid* is a short integer value specifying the communication device to carry out the extended function.

nFunc is a short integer value specifying the function code of the extended function. It can be any one of the following:

Value	Meaning
SETXOFF	Causes transmission to behave as if an XOFF character has been received.
SETXON	Causes transmission to behave as if an XON character has been received.
SETRTS	Asserts the Request To Send (RTS) signal.
CLRRTS	Clears the Request To Send (RTS) signal.
SETDTR	Asserts the Data Terminal Ready (DTR) signal.
CLRDTR	Clears the Data Terminal Ready (DTR) signal.
RESETDEV	Reset the device if possible.

Return Value *nResult* is a short integer value specifying the result of the function. It is 0 if it is successful. It is negative if *nFunc* is not a valid function code.

4.11 Sound Functions

This section describes the functions used to create sounds and music for the system's sound generator.

OpenSound () : nVoices

Purpose This function opens access to the play device and prevents subsequent opening of the device by other applications.

Parameters None.

Return Value *nVoices* is a short integer value specifying the number of voices available. It is S_SERDVNA if the play device is in use. It is S_EROFM if insufficient memory is available.

CloseSound ()

Purpose This function closes access to the play device and frees the device for opening by other applications. **CloseSound** flushes all voice queues and frees any buffers allocated for these queues.

Parameters None.

Return Value None.

SetVoiceQueueSize (nVoice, nBytes) : nResult

Purpose This function allocates *nBytes* bytes for the voice queue specified by *nVoice*. If the queue size is not set, the default is 192 bytes, room for about 32 notes. All voice queues are locked in memory. The queues cannot be set while music is playing.

Parameters *nVoice* is a short integer value specifying a voice queue.

nBytes is a short integer value specifying the number of bytes in the voice queue.

Return Value *nResult* is a short integer value specifying the result of the function. It is zero if the function is successful. On a error, it is one of the following:

S_SEROFM	Out of memory
S_SERMACT	Music active

SetVoiceAccent (*nVoice*, *nTempo*, *nVolume*, *nMode*, *nPitch*) : *nResult*

Purpose This function places an accent (tempo, volume, mode, and pitch) in the voice queue *nVoice*. The new accent replaces the previous accent and remains in effect until another accent is queued. An accent is not counted as a note.

An error occurs if there is insufficient room in the queue; **SetVoiceAccent** always leaves space for a single sync mark in the queue. If *nVoice* is out of range, **SetVoiceAccent** is ignored.

Parameters *nVoice* is a short integer value specifying a voice queue. The first voice queue is numbered 1.

nTempo is a short integer value specifying the number of quarter notes played per minute. It can be any value in the range 32 to 255. The default is 120.

nVolume is a short integer value specifying the volume level. It can be any value in the range 0 (lowest volume) to 255 (highest volume).

nMode is a sort integer value specifying how the notes are to be played. It can be any one of the following:

Value	Meaning
S_LEGATO	Note is held for the full duration and blended with the beginning of the next.
S_STACCATO	Note is held for only part of the duration, creating a pronounced stop between it and the next note.
S_NORMAL	Note is held for the full duration, coming to a full stop before the next note starts.

nPitch is a short integer value specifying the pitch of the notes to be played. It can be any value in the range 0 to 83. The pitch value is added to the note value using modulo 84 arithmetic.

Return Value *nResult* is a short integer value specifying the result of the function. It is zero if the function is successful. On an error it is one of the following:

Value	Error
S_SERQFUL	Queue full
S_SERDVL	Invalid volume
S_SERDTP	Invalid tempo
S_SERDMD	Invalid mode

SetVoiceEnvelope (*nVoice*, *nShape*, *nRepeat*) : *nResult*

Purpose This function queues the envelope (wave shape and repeat count) in the voice queue specified by *nVoice*. The new envelope replaces the previous one and remains in effect until the next **SetVoiceEnvelope** call. An envelope is not counted as a note.

An error occurs if there is insufficient room in the queue; **SetVoiceEnvelope** always leaves space for a single sync mark in the queue. If *nVoice* is out of range, **SetVoiceEnvelope** is ignored.

Parameters *nVoice* is a short integer value specifying the voice queue to receive the envelope.

nShape is a short integer value specifying an index to an OEM wave shape table.

nRepeat is a short integer specifying the number of repetitions of the shape during the duration of one note.

Return Value *nResult* is a short integer value specifying the result of the function. It is zero if the function is successful. On an error, it is one of the following:

Value	Error
S_SERQFUL	Queue full
S_SERDSH	Invalid shape
S_SERDRC	Invalid repeat count

SetSoundNoise (*nSource*, *nDuration*) : *nResult*

Purpose This function sets the source and duration of a noise in the noise hardware of the play device.

Parameters *nSource* is a short integer value specifying the noise source. It can be any one of the following values:

Value	Meaning
S_PERIOD512	Source frequency is N/512 (high pitch), hiss is less coarse
S_PERIOD1024	Source frequency is N/1024
S_PERIOD2048	Source frequency is N/2048 (low pitch), hiss is coarser
S_PERIODVOICE	Source frequency from voice channel 3
S_WHITE512	Source frequency is N/512 (high pitch), hiss is less coarse
S_WHITE1024	Source frequency is N/1024
S_WHITE2048	Source frequency is N/2048 (low pitch), hiss is coarser
S_WHITEVOICE	Source frequency from voice channel 3

nDuration is a short integer value specifying the duration of the noise in clock tics.

Return Value *nResult* is a short integer value specifying the result of the function. It is zero if the function is successful. If the source is invalid, it is S_SERDSR.

SetVoiceNote (*nVoice*, *nValue*, *nLength*, *nCdots*) : *nResult*

Purpose This function queues a note with the qualities given by *nValue*, *nLength*, and *nCdots* in the voice queue specified by *nVoice*. An error occurs if there is insufficient room in the queue. The function always leaves space in the queue for a single sync mark.

Parameters *nVoice* is a short integer value specifying the voice queue to receive the note. If *nVoice* is out of range, **SetVoiceNote** is ignored.

nValue is a short integer value specifying one of 84 possible notes (7 octaves). If *nValue* is zero, a rest is assumed.

nLength is a short integer value specifying the reciprocal of the duration of the note. For example, 1 specifies a whole note, 2 a half note, 4 a quarter note, and so on.

nCdots is a sort integer value specifying the duration of the note in dots. The duration is equal to $nLength*(nCdots*3/2)$.

Return Value *nResult* is a short integer value specifying the result of the function. It is zero if the function is successful. On an error, it is one of the following:

Value	Error
S_SERQFUL	Queue full
S_SERDNT	Invalid note
S_SERDLN	Invalid note length
S_SERDCC	Invalid Cdot count

SetVoiceSound (*nVoice*, *lFrequency*, *nDuration*) : *nResult*

Purpose This function queues the sound frequency and duration in the voice queue specified by *nVoice*.

Parameters *nVoice* is a short integer value specifying a voice queue. The first voice queue is numbered 1.

lFrequency is a long integer specifying the frequency. The high-order word contains the frequency in Kilohertz, and the low-order word contains the fractional frequency.

nDuration is a short integer specifying the duration of the sound in clock ticks.

Return Value *nResult* is a short integer value specifying the result of the function. It is zero if the function is successful. On an error, it is one of the following:

Value	Error
S_SERQFUL	Queue full
S_SERDFQ	Invalid frequency
S_SERDDR	Invalid duration
S_SERDVL	Invalid volume

StartSound() : nResult

Purpose This function starts play in each voice queue. The **Start-Sound** function is not destructive, so may be called any number of times to replay the current queues.

Parameters None.

Return Value *nResult* is a short integer.

StopSound() : nResult

Purpose This function stops playing all voice queues, then flushes the contents of the queues. The sound driver for each voice is turned off.

Parameters None.

Return Value *nResult* is a short integer.

WaitSoundState(nState) : nResult

Purpose This function waits until the play driver enters the specified state.

Parameters *nState* is a short integer value specifying the state of the voice queues. It can be any one of the following:

Value	State
S_QUEUEEMPTY	All voice queues are empty and sound drivers turned off
S_THRESHOLD	A voice queue has reached threshold, returns voice
S_ALLTHRESHOLD	All voices have reached threshold

Return Value *nResult* is a short integer value specifying the result of the function. It is zero if the function is successful. If the state is not valid, it is S_SERDST.

SyncAllVoices () : nResult

Purpose This function queues a sync mark in each queue. Upon encountering a sync mark in a voice queue, the voice is turned off until sync marks are encountered in all other queues. This forces synchronization between all voices.

Parameters None.

Return Value *nResult* is a short integer value specifying the result of the function. It is zero if the function is successful. If a voice queue is full, *nResult* is S_SERQFUL.

CountVoiceNotes (*nVoice*) : *nNotes*

Purpose This function retrieves a count of the number of notes in the specified queue. Only those queue entries resulting from calls to **SetVoiceNote** are counted.

Parameters *nVoice* is a short integer value specifying the voice queue to be counted. The first voice queue is numbered 1.

Return Value *nNotes* is a short integer number specifying the number of notes in the given queue.

GetThresholdEvent () : *lpInt*

Purpose This function retrieves a flag identifying a recent threshold event. A threshold event is any transition of a voice queue from *n* to *n-1* where *n* is the threshold level in notes.

Parameters None.

Return Value *lpInt* is a long pointer to a short integer specifying a threshold event.

GetThresholdStatus () : *fStatus*

Purpose This function retrieves the threshold event status for each voice. Each bit in the status represents a voice. If a bit is set, the voice queue level is currently below threshold.

The **GetThresholdStatus** function also clears the threshold event flag. The threshold flag should be cleared only in this way.

Parameters None.

Return Value *fStatus* is a short integer containing the status flags of the current threshold event.

SetVoiceThreshold (*nVoice*, *nNotes*) : *nResult*

Purpose This function sets the threshold level for the given voice. When the number of notes remaining in the voice queue goes below *nNotes* the threshold flag is set. If the queue level is below *nNotes* when **SetVoiceThreshold** is called, the flag is not set. The **GetThresholdStatus** function should be called to verify the current threshold status.

Parameters *nVoice* is a short integer value specifying the voice queue to be set.

nNotes is a short integer number specifying the number of notes in the threshold level.

Return Value *nResult* is a short integer value specifying the result of the function. It is zero if the function is successful. It is 1 if *nNotes* is out of range.

4.12 Utility Functions

This section describes useful functions and macros.

HIBYTE (*nInteger*) : *cHighByte*

Purpose This function retrieves the high-order byte from the integer value *nInteger*.

Parameters *nInteger* is a short integer value.

Return Value *cHighByte*, a byte integer value, is the high-order byte of the given value.

LOBYTE (*nInteger*) : *cLowByte*

Purpose This function extracts the low-order byte from the short integer value *nInteger*.

Parameters *nInteger* is a 16-bit integer value.

Return Value *cLowByte*, a byte integer value, is the low-order byte of the value.

HIWORD (*lInteger*) : *wHighWord*

Purpose This function retrieves the high-order word from the long integer value *lInteger*.

Parameters *lInteger* is a long integer value.

Return Value *wHighWord*, an unsigned short integer value, is the high-order word of the given long value.

LOWORD (*lInteger*) : *wLowWord*

Purpose This function extracts the low-order word from the long value *lInteger*.

Parameters *lInteger* is a long integer value.

Return Value *wLowWord*, an unsigned short integer value, is the low-order word of the long value.

MAKELONG (*nLowWord*, *nHighWord*) : *dwInteger*

Purpose This function creates a unsigned long integer by concatenating two integer values, *nLowWord* and *nHighWord*.

Parameters *nLowWord* is a short integer value specifying the low-order word of the new long value.

nHighWord is a short integer value specifying the high-order word of the new long value.

Return Value *dwInteger* is an unsigned long integer value.

MAKEINTRESOURCE (*nInteger*) : *lpString*

Purpose This function converts an integer value to a long pointer to a string (LPSTR type), with the high-order word of the long pointer set to zero.

Parameters *nInteger* is a short integer value.

Return Value *lpString* is a long pointer to a string (LPSTR type).

MAKEPOINT (*lInteger*) : Point

Purpose This macro converts a long value containing the x and y coordinates of a point into a **POINT** structure.

Parameters *lInteger* is a long value containing the x coordinate of the point in the low-order word and the y coordinate is in the high-order word.

Return Value *Point* is a structure with **POINT** type containing the x and y coordinates.

GetRValue (*rgbColor*) : *cRedValue*

Purpose This macro extracts the red value from an RGB color value.

Parameters *rgbColor* is an RGB color value.

Return Value *cRedValue* is a byte containing the red value of *rgbColor*.

GetGValue (*rgbColor*) : *cGreenValue*

Purpose This macro extracts the green value from an RGB color value.

Parameters *rgbColor* is an RGB color value.

Return Value *cGreenValue* is a byte containing the green value of *rgbColor*.

GetBValue (*rgbColor*) : *cBlueValue*

Purpose This macro extracts the blue value from an RGB color value.

Parameters *rgbColor* is an RGB color value.

Return Value *cBlueValue* is a byte containing the blue value of *rgbColor*.

min (*Value1*, *Value2*) : *MinValue*

Purpose This macro returns the minimum of *Value1* and *Value2*.

Parameters *Value1* and *Value2* are two arbitrary values to be compared.

Return Value The return value is *Value1* or *Value2*, whichever is lesser.

max (Value1, Value2) : MaxValue

Purpose This macro returns the maximum of *Value1* and *Value2*.

Parameters *Value1* and *Value2* are two arbitrary values to be compared.

Return Value The return value is *Value1* or *Value2*, whichever is greater.

4.13 File I/O Functions

This section describes functions used for creating and opening files.

OpenFile (lpFileName, lpReOpenBuff, wStyle) : hFile

Purpose This function creates, opens, reopens, or deletes a file.

Parameters *lpFileName* is a long pointer to a null-terminated character string specifying the name of the file to be opened. The string must consist of characters from the ANSI character set.

lpReOpenBuff is a long pointer to a data structure having **OFSTRUCT** type. The structure receives information about the file when the file is first opened, and is used in subsequent **OpenFile** calls to refer to the open file.

wStyle is an unsigned short integer value specifying the action to take. It can be combinations of the following:

Value	Meaning
OF_READ	Opens the file for reading only.
OF_WRITE	Opens the file for writing only.
OF_READWRITE	Opens the file for reading and writing.
OF_CREATE	Creates the file if it does not already exist.
OF_REOPEN	Opens the file using information in the reopen buffer.
OF_EXIST	Creates or opens the file, then closes it. Used to test for file existence.

OF_PARSE	Fills the OFSTRUCT structure but carries out no other action.
OF_PROMPT	Displays a dialog box that prompts the user for permission to create a file if the requested file does not exist.
OF_CANCEL	Adds a Cancel button to the OF_PROMPT dialog box. Pressing the Cancel button directs OpenFile to return a file-not-found error.
OF_VERIFY	Verifies that the date and time of the file are the same as when it was previously opened. Useful as an extra check for read-only files.
OF_DELETE	Deletes the file.

These styles can be combined using the logical OR operator.

Return Value *hFile* is an MS-DOS file handle if the function is successful. Otherwise, *hFile* is -1.

Notes To close the file after use, use the MS-DOS system close call.

GetTempFileName (*cDriveLetter*, *lpPrefixString*, *wUnique*, *lpTempFileName*) : *nUniqueNumber*

Purpose This function creates a temporary filename with the following form:

drive:\ path\ ~prefixuuuu.TMP

where *drive* is the drive letter specified by *cDriveLetter*, *path* is the pathname of the temporary file (either the root directory of the specified drive or the directory specified in the TEMP environment variable), *prefix* is all letters (up to the first three) of the string pointed to by *lpPrefixString*, and *uuuu* is a hexadecimal representation of the number specified by *wUnique*.

<i>Parameters</i>	<p><i>cDriveLetter</i> is a byte integer specifying the suggested drive for the temporary filename. If <i>cDriveLetter</i> is 0, the default drive is used.</p> <p><i>lpPrefixString</i> is a long pointer to a null-terminated string to be used as the temporary filename prefix.</p> <p><i>wUnique</i> is an unsigned short integer.</p> <p><i>lpTempFileName</i> is a long pointer to a buffer where the temporary filename is stored.</p>
<i>Return Value</i>	<p><i>nUniqueNumber</i>, a short integer value, is the unique numeric value used in the temporary filename. If a nonzero value was given for the <i>wUnique</i> parameter, <i>nUniqueNumber</i> is the same number.</p>
<i>Notes</i>	<p>GetTempFileName uses the suggested drive letter for creating the temporary filename except in the following cases:</p> <ol style="list-style-type: none">1. If a hard disk is present, GetTempFileName always uses the drive letter of the first hard disk.2. If a TEMP environment variable is defined and its value begins with a drive letter, that drive letter is used. <p>If the TF_FORCECREATE bit of <i>cDriveLetter</i> is set, the above exceptions do not apply. The temporary filename will always be created in the current directory of the drive specified by <i>cDriveLetter</i>, regardless of the presence of a hard disk or the TEMP environment variable.</p> <p>If <i>wUnique</i> is zero, GetTempFileName attempts to form a unique number based on the current system time. If a file with the resulting filename already exists, the number is increased by one and the test for existence is repeated. This process continues until a unique filename is found; GetTempFileName then creates a file by that name and closes it.</p> <p>No attempt is made to create and open the file when <i>wUnique</i> is nonzero.</p>

GetTempDrive (*cDriveLetter*) : *cOptDriveLetter*

Purpose This function takes a drive letter or zero and returns a letter specifying the optimal drive for a temporary file. The optimal drive is the disk drive that can provide the best access time during disk operations with a temporary file.

The function returns the drive letter of a hard disk if the system has one. Otherwise, if *cDriveLetter* is zero, it returns the drive letter of the current disk, or if *cDriveLetter* is a letter, it returns the letter of that drive or one that exists.

Parameters *cDriveLetter* is a byte integer value specifying a disk drive letter, for example, "A" for disk drive A.

Return Value *cOptDriveLetter* is a byte integer value specifying the optimal disk drive for temporary files.

Chapter 5

Data Types and Structures

- 5.1 Introduction 251
- 5.2 Data Types 251
- 5.3 Window Data Structures 254
- 5.4 GDI Data Structures 259
- 5.5 Communication Data Structures 272
- 5.6 Open File Structure 277

(

)

)

5.1 Introduction

This chapter describes the data types and structures used by Windows functions.

5.2 Data Types

This section describes the data types used with Windows functions. The data types are keywords that define the size and meaning of the parameters and return values associated with the functions.

Value Types

Windows functions use the following character, integer, and Boolean types:

Type	Definition
char	Specifies an ASCII character, or a signed 8-bit integer.
BYTE	Specifies an unsigned 8-bit integer.
int	Specifies a signed 16-bit integer.
short	Specifies a signed 16-bit integer.
WORD	Specifies an unsigned 16-bit integer.
long	Specifies a signed 32-bit integer.
LONG	Specifies a signed 32-bit integer.
DWORD	Specifies an unsigned 32-bit integer, or a segment/offset address combination.
BOOL	Specifies a 16-bit Boolean value.
VOID	Specifies an empty value. It is used with a function to specify no return value.

Pointer Types

The Windows functions use many pointer types. Most pointer type names begin with either a "P" prefix for a short pointer (i.e., data is within the current data segment), or "LP" for a long pointer (i.e., a 32-bit segment/offset pointer). The following is a list of common pointer types:

Type	Definition
PSTR	Specifies a pointer to a character string.
PINT	Specifies a pointer to a signed 16-bit integer.
LPSTR	Specifies a long pointer to a character string.
LPINT	Specifies a long pointer to a signed 16-bit integer.
LPRECT	Specifies a long pointer to a RECT data structure.
LPMSG	Specifies a long pointer to a MSG data structure.
FARPROC	Specifies a long pointer to a function.
FAR	Specifies a data type attribute that can be used to create a long pointer.
NEAR	Specifies a data type attribute that can be used to create a short pointer.

Handles

Windows uses a large variety of handles to refer to resources that have been loaded into memory and can be used by applications. Windows provides access to resources through internally maintained tables. The entries in the table define the locations and type of data available. The handle is an index to the table.

The following is a list of common handles:

Type	Definition
HANDLE	Specifies a general handle. It represents a 16-bit index to a table entry identifying program data.
HSTR	Specifies a handle to a string resource. It is a 16-bit index to a resource table entry.

HCURSOR	Specifies a handle to a cursor resource. It is a 16-bit index to a resource table entry.
HICON	Specifies a handle to an icon resource. It is a 16-bit index to a resource table entry.
HMENU	Specifies a handle to a menu resource. It is a 16-bit index to a resource table entry.
HDC	Specifies a handle to a display context. It is a 16-bit index to GDI's display context tables.
HPEN	Specifies a handle to a physical pen. It is a 16-bit index to GDI's physical drawing objects.
HFONT	Specifies a handle to a physical font. It is a 16-bit index to GDI's physical drawing objects.
HBRUSH	Specifies a handle to a physical brush. It is a 16-bit index to GDI's physical drawing objects.
HBITMAP	Specifies a handle to a physical bitmap. It is a 16-bit index to GDI's physical drawing objects.
HRGN	Specifies a handle to a physical region. It is a 16-bit index to GDI's physical drawing objects.
GLOBALHANDLE	Specifies a handle to global memory. It is a 16-bit index to a block of memory allocated from the system's global heap.
LOCALHANDLE	Specifies a handle to local memory. It is a 16-bit index to a block of memory allocated from the application's local heap.

5.3 Window Data Structures

This section describes the data structures used for window class registration, application queue messages, window messages, and window creation.

WNDCLASS - *Window Class Data Structure*

The **WNDCLASS** structure contains the following fields:

WORD	style
FARPROC	lpfnWndProc
int	cbClsExtra
int	cbWndExtra
HANDLE	hInstance
HICON	hIcon
HCURSOR	hCursor
HBRUSH	hbrBackground
LPSTR	lpszMenuName
LPSTR	lpszClassName

The **WNDCLASS** fields have the following meanings:

Field	Definition	
style	Specifies the class style. It can be any combination of the following:	
	CS_VREDRAW	Redraw entire window if vertical size changes.
	CS_HREDRAW	Redraw entire window if horizontal size changes.
	CS_KEYCWTWINDOW	Reserve space at bottom of screen for key conversion window.
	CS_DBCLKS	Send double click messages to window.
	CS_OEMCHARS	OEM character translation (not used by Windows applications).

	CS_OWNDC	Give each window instance its own display context.
	CS_CLASSDC	Give the window class its own display context (shared by instances).
	<i>Note</i>	Although the CS_OWNDC style is convenient, each display context occupies approximately 800 bytes of memory, so it must be used with discretion.
		The above styles can be combined using the bitwise OR operator.
	lpfnWndProc	Specifies the window function. It must be a long pointer to the function.
	cbClsExtra	Specifies the number of bytes to allocate after the window class structure.
	cbWndExtra	Specifies the number of bytes to allocate after the window instance.
	hInstance	Specifies the class module. It must be an instance handle. It must not be NULL.
	hCursor	Specifies the class cursor. It must be a handle to a cursor resource. If <i>hCursor</i> is NULL, the application must explicitly set the cursor shape whenever the mouse moves into the application's window.
	hIcon	Specifies the class icon. It must be a handle to an icon resource. If <i>hIcon</i> is NULL, the application must draw an icon whenever the user closes the application's window.

hbrBackground Specifies the class background brush. It can be either a handle to the physical brush to be used for painting the background or a color value. If a color value is given, it must be one of the standard system colors listed below, and the value 1 must be added to the chosen color (for example, COLOR_BACKGROUND+1 specifies the system background color). If a color value is given, it must be converted to **HBRUSH** type.

COLOR_SCROLLBAR
COLOR_BACKGROUND
COLOR_ACTIVECAPTION
COLOR_INACTIVECAPTION
COLOR_MENU
COLOR_WINDOW
COLOR_WINDOWFRAME
COLOR_MENUTEXT
COLOR_WINDOWTEXT
COLOR_CAPTIONTEXT

When *hbrBackground* is NULL, the application must paint its own background whenever it is requested to paint in its client area. The application can determine when the background needs painting by handling the WM_ERASEBKND message or by testing the *fErase* field of the **PAINTSTRUCT** structure filled by **BeginPaint**.

lpszMenuName Specifies the resource name of the class menu (as the name appears in the resource file). It must be a long pointer to a null-terminated ASCII string. If an integer value is used to identify the menu, the **MAKEINTRESOURCE** macro can be used. If *lpszMenuName* is NULL, windows belonging to this class have no default menu.

lpszClassName Specifies the name of the window class. It must be a long pointer to a null-terminated ASCII string.

MSG - Message Data Structure

The **MSG** structure contains information from the Windows application queue. The structure has the following fields:

HWND	hwnd
WORD	message
WORD	wParam
LONG	lParam
DWORD	time
POINT	pt

The **MSG** fields have the following meanings:

Field	Definition
hwnd	Specifies a handle to the window receiving the message.
message	Specifies the message number.
wParam	Specifies additional information about the message. The exact meaning depends on the <i>message</i> value.
lParam	Specifies additional information about the message. The exact meaning depends on the <i>message</i> value.
time	Specifies the time at which the message was posted.
pt	Specifies the position of the mouse (in screen coordinates) when the message was posted.

PAINTSTRUCT - Windows Paint Information

The **PAINTSTRUCT** structure contains information for an application that can be used to paint the client area of a window owned by the application. The structure has the following fields:

HDC	hdc
BOOL	fErase
RECT	rcPaint
BOOL	fRestore
BOOL	fIncUpdate
BYTE	rgbReserved[16]

The PAINTSTRUCT fields have the following meanings:

Field	Definition
hdc	Specifies the display context to be used for painting.
fErase	Specifies whether or not the background has been redrawn. It has been redrawn if nonzero.
rcPaint	Specifies the upper left and lower right corners of the rectangle in which the painting is requested.
fRestore	Used internally by Windows.
fIncUpdate	Used internally by Windows.
rgbReserved	Specifies a reserved block of memory used internally by Windows.

CREATESTRUCT - Window Creation Structure

The **CREATESTRUCT** structure defines the initialization parameters passed to an application's window function with the WM_CREATE and WM_NCCREATE messages. The structure contains the following fields:

LPSTR	lpCreateParams
HANDLE	hInstance
HANDLE	hMenu
HWND	hwndParent
int	cy
int	cx
int	y
int	x
long	style
LPSTR	lpszName
LPSTR	lpszClass

The **CREATESTRUCT** fields have the following meanings:

Field	Definition
lpCreateParams	Specifies the window creation parameters. It can be a long pointer to data to be used when creating the window.
hInstance	Specifies the module instance handle of the module owning the new window.
hMenu	Specifies the handle of the menu to be used by the new window.

hwndParent	Specifies the window handle of the window owning the new window. It is NULL if the new window is a top-level window.
cy	Specifies the height of the new window.
cx	Specifies the width column of the new window. It is used by child and popup windows only.
y	Specifies the y coordinate of the upper left corner of the new window. Coordinates are relative to the parent window if the new window is a child. Otherwise, they are relative to the screen origin.
x	Specifies the x coordinate of the upper left corner of the new window. Coordinates are relative to the parent window if the new window is a child. Otherwise, they are relative to the screen origin.
style	Specifies the new window's style.
lpszName	Specifies the new window's name. It is a long pointer to a null-terminated ASCII string.
lpszClass	Specifies the new window's class name. It is a long pointer to a null-terminated ASCII string.

5.4 GDI Data Structures

This section describes the data structures used by the Graphics Device Interface (GDI) functions.

LOGPEN - Logical Pen Attribute Information

The **LOGPEN** structure defines the style, width, and color of a pen. A pen is a drawing object used to draw lines and borders. The **LOGPEN** structure is used in the **CreatePenIndirect** function.

The structure contains the following fields:

WORD	lpenStyle
POINT	lpenWidth
DWORD	lpenColor

The **LOGOPEN** fields have the following meanings:

Field	Definition
lopnStyle	Specifies the pen type. It can be any one of the following:
0	Solid
1	Dashed
2	Dotted
3	Dash-dotted
4	Dash-dot-dotted
5	Null
lopnWidth	Specifies the pen width in logical units. If <i>lopnWidth</i> is 0, the pen is one pixel width on raster devices.
lopnColor	Specifies the pen color. It must be an RGB color value.

LOGBRUSH - Logical Brush Attribute Information

The **LOGBRUSH** structure defines the style, color, and pattern of a physical brush to be created using the **CreateBrushIndirect** function. The structure contains the following fields:

WORD	lbStyle
DWORD	lbColor
short	lbHatch

The **LOGBRUSH** fields have the following meanings:

Field	Definition
lbStyle	Specifies the brush style. It can be any one of the following:
	BS_SOLID BS_HOLLOW BS_HATCHED BS_PATTERN
lbColor	Specifies the color in which the brush is to be drawn. It must be an RGB color value. If <i>lbStyle</i> is BS_HOLLOW or BS_PATTERN, <i>lbColor</i> is ignored.

lbHatch Specifies a hatch style. The meaning depends on the brush style.

If *lbStyle* is BS_HATCHED, *lbHatch* specifies the orientation of the lines used to create the hatch. It can be any one of the following:

HS_HORIZONTAL - horizontal hatch

HS_VERTICAL - vertical hatch

HS_FDIAGONAL - 45-degree upward hatch from left to right

HS_BDIAGONAL - 45-degree downward hatch from left to right

HS_CROSS - horizontal and vertical cross-hatch

HS_DIAGCROSS - 45-degree cross-hatch

If *lbStyle* is BS_PATTERN, *lbHatch* must be a handle to the bitmap defining the pattern.

If *lbStyle* is BS_SOLID or BS_HOLLOW, *lbHatch* is ignored.

LOGFONT *Logical Font Descriptor*

The **LOGFONT** structure defines the attributes of a font. A font is a drawing object used to write text on a display surface. The structure is used in the **CreateFontIndirect** function to direct GDI to select a matching physical font. The **LOGFONT** structure contains the following fields:

short	lfHeight
short	lfWidth
short	lfEscapement
short	lfOrientation
short	lfWeight
BYTE	lfItalic
BYTE	lfUnderline
BYTE	lfStrikeOut
BYTE	lfCharSet
BYTE	lfOutPrecision
BYTE	lfClipPrecision
BYTE	lfQuality
BYTE	lfPitchAndFamily
BYTE	lfFaceName[LF_FACESIZE]

The **LOGFONT** fields have the following meanings:

Field	Definition
lfHeight	Specifies the height of the font in user units. The height of a font can be specified in three ways. If <i>lfHeight</i> is greater than zero, it is transformed into device units and matched against the cell height of the available fonts. If <i>lfHeight</i> is zero, a reasonable default size is used. If <i>lfHeight</i> is less than zero, it is transformed into device units and the absolute value is matched against the character height of the available fonts.
lfWidth	Specifies the average width of characters in the font in user units. If <i>lfWidth</i> is zero, the aspect ratio of the device will be matched against the digitization aspect ratio of the available fonts looking for the closest match by absolute value of the difference.
lfEscapement	Specifies the angle in tenths of degrees between the escapement vector and the x-axis of the display surface. The escapement vector is the line through the origins of the first and last characters on a line. The angle is measured counterclockwise from the x-axis.
lfOrientation	Specifies the angle in tenths of degrees between the baseline of a character and the x-axis. The angle is measured counterclockwise from the x-axis.
lfWeight	Specifies the font weight in inked pixels per 1000. Although it can be any integer value from 0 to 1000, the common values are: 400 normal 700 bold These values are approximate; the actual appearance depends on the font face. If <i>lfWeight</i> is 0, a default weight is used.
lfItalic	Indicates an italic font if set to nonzero.
lfUnderline	Indicates an underlined font if set to nonzero.
lfStrikeOut	Indicates a strikeout font if set to nonzero.

lfCharSet Specifies the font's character set. It can be any one of the following:

ANSI_CHARSET
OEM_CHARSET

The characters of the ANSI_CHARSET set are defined in Appendix C. The OEM_CHARSET set is system-dependent.

lfOutPrecision Specifies the font's output precision. The output precision defines how closely the output must match the requested font's height, width, character orientation, escapement, and pitch. Currently must be set to OUT_DEFAULT_PRECIS.

lfClipPrecision Specifies the font's clipping precision. The clipping precision defines how to clip characters that are partially outside the clipping region. Currently must be set to CLIP_DEFAULT_PRECIS.

lfQuality Specifies the font's output quality. The output quality defines how carefully GDI must attempt to match the logical font attributes to those of an actual physical font. It can be any one of the following:

PROOF_QUALITY

The character quality of the font is more important than exact matching of the logical font attributes. For GDI fonts, scaling is disabled and the font closest in size is chosen. Although the chosen font size may not be mapped exactly when PROOF_QUALITY is used, the quality of the font is high and there is no degradation of appearance. Bold, italic, underline, and strikeout are synthesized if needed.

DRAFT_QUALITY

The appearance of the font is less important than when PROOF_QUALITY is used. For GDI fonts, scaling is enabled, with the result that more font sizes are available, but the quality may be lower. Bold, italic, underline, and strikeout are synthesized if needed.

DEFAULT_QUALITY

The appearance of the font does not matter.

lfPitchAndFamily Specifies the font pitch and family. The two low-order bits specify the pitch of the font and can be any one of the following:

DEFAULT_PITCH
FIXED_PITCH
VARIABLE_PITCH

The four high-order bits of the field specify the font family and can be any one of the following:

FF_DONTCARE
FF_ROMAN
FF_SWISS
FF_MODERN
FF_SCRIPT
FF_DECORATIVE

The constants are defined such that the proper value can be obtained by ORing together one pitch constant with one family constant. Font families describe the look of a font in a general way. They are intended for specifying fonts when the exact facename desired is not available. The families are:

FF_DONTCARE
Don't care or don't know.

FF_ROMAN
Fonts with variable stroke width, serifed.
Times Roman, Century Schoolbook, etc.

FF_SWISS
Fonts with variable stroke width, sans-serifed.
Helvetica, Swiss, etc.

FF_MODERN
Fonts with constant stroke width, serifed
or sans-serifed. Fixed-pitch fonts are
usually modern. Pica, Elite, Courier, etc.

FF_SCRIPT
Cursive, etc.

FF_DECORATIVE
Old English, etc.

lfFaceName Specifies the name of the font's typeface. It must be a null-terminated ASCII character string. If *lfFaceName* is NULL, GDI uses a default face.

BITMAP - *Bitmap Data Structure*

The **BITMAP** structure defines the height, width, color format, and bit values of a logical bitmap. This structure can be used to create a bitmap with the **CreateBitmapIndirect** function and can be filled in for a particular bitmap by passing the bitmap handle to **GetObject**.

The **BITMAP** structure contains the following fields:

```
short    bmType
short    bmWidth
short    bmHeight
short    bmWidthBytes
BYTE     bmPlanes
BYTE     bmBitsPixel
LPSTR   bmBits
```

The fields within the **BITMAP** structure have the following meanings:

Field	Definition
bmType	Specifies the bitmap type. For logical bitmaps, this must be 0.
bmWidth	Specifies the width of the bitmap in pixels. The width must be greater than 0.
bmHeight	Specifies the height of the bitmap in raster lines. The height must be greater than 0.
bmWidthBytes	Specifies the number of bytes in each raster line. This must be an even number since GDI assumes that the bit values of a bitmap form an array of integer (2-byte) values. In other words, <i>bmWidthBytes</i> *8 must be the next multiple of eight greater than or equal to <i>bmWidth</i> .
bmPlanes	Specifies the number of color planes in the bitmap.
bmBitsPixel	Specifies the number of adjacent color bits on each plane needed to define a pixel.
bmBits	Specifies the location of the bit values for the bitmap. It must be a long pointer to an array of character (1-byte) values.

POINT - Point Data Structure

The **POINT** structure defines the x and y coordinates of a point. The structure contains the following fields:

```
int    x  
int    y
```

The fields have the following meanings:

Field	Definition
x	is the x coordinate value of a point.
y	is the y coordinate value of a point.

RECT - Rectangle Data Structure

The **RECT** structure defines the coordinates of the upper left and lower right corners of a rectangle. The structure contains the following fields:

```
int    left  
int    top  
int    right  
int    bottom
```

The fields have the following meanings:

Field	Definition
left	Specifies the x coordinate of the upper left corner of a rectangle.
top	Specifies the y coordinate of the upper left corner of a rectangle.
right	Specifies the x coordinate of the lower right corner of a rectangle.
bottom	Specifies the y coordinate of the lower right corner of a rectangle.

The width of the rectangle defined by the **RECT** structure must not exceed 32K units.

RGB - Logical Color Specification

A RGB color value is a long integer containing a red, a green, and a blue color field, each specifying the intensity of the given color. Intensities range from 0 to 255. The values are packed in the three low-order bytes of the long integer in the following format: r + 2⁸*g + 2¹⁶*b.

0FF (hexadecimal) is the maximum value for a single byte, and 0 is the minimum. Black (that is, no color information) corresponds to 0, 0, 0 (0 in all three bytes). White is all color: FF, FF, FF. Gray is half energy, corresponding to 7F, 7F, 7F. Solid green would be 00, FF, 00.

TEXTMETRIC - Basic Font Metrics

The **TEXTMETRIC** structure is a list of the basic metrics of a physical font. The structure is returned by the **GetTextMetrics** function.

The **TEXTMETRIC** structure contains the following fields:

short	tmHeight
short	tmAscent
short	tmDescent
short	tmInternalLeading
short	tmExternalLeading
short	tmAveCharWidth
short	tmMaxCharWidth
short	tmWeight
BYTE	tmItalic
BYTE	tmUnderlined
BYTE	tmStruckOut
BYTE	tmFirstChar
BYTE	tmLastChar
BYTE	tmDefaultChar
BYTE	tmBreakChar
BYTE	tmPitchAndFamily
BYTE	tmCharSet
short	tmOverhang
short	tmDigitizedAspectX
short	tmDigitizedAspectY

The **TEXTMETRIC** fields are described below. All sizes are given in logical units (i.e., they depend on the current mapping mode of the display context).

Field	Definition
tmHeight	Specifies the height of characters (Ascent + Descent).
tmAscent	Specifies the ascent of characters (units above the baseline).
tmDescent	Specifies the descent of characters (units below the baseline).
tmInternalLeading	Specifies the amount of leading inside the bounds set by <i>tmHeight</i> . Accent marks may occur in this area. This may be zero at the designer's option.
tmExternalLeading	Specifies the amount of extra leading that the designer requests the application add between rows. Since this area is outside the font, it contains no marks, and will not be altered by text output calls in either OPAQUE or TRANSPARENT mode. This may be zero at the designer's option.
tmAveCharWidth	Specifies the average width of characters in the font (loosely defined as the width of the letter 'X').
tmMaxCharWidth	Specifies the maximum width of any character in the font.
tmWeight	Specifies the weight of the font.
tmItalic	Specifies an italic font if it is nonzero.
tmUnderlined	Specifies an underlined font if it is nonzero.
tmStruckOut	Specifies an struckout font if it is nonzero.
tmFirstChar	Specifies the value of the first character defined in the font.
tmLastChar	Specifies the value of the last character defined in the font.
tmDefaultChar	Specifies the value of the character that will be substituted for characters that are not in the font.

tmBreakChar	Specifies the value of the character that will be used to define word breaks for text justification.
tmPitchAndFamily	Specifies the pitch and family of the selected font. The low-order bit is set if the font is variable pitch. The four high-order bits give the family of the font. <i>tmPitchAndFamily</i> can be combined with the hexadecimal value F0, using the bitwise AND operator, and then directly compared for equality with the font family names. Refer to the LOGFONT structure for a description of the font families.
tmCharSet	Specifies the character set of the font.
tmOverhang	Specifies the per string extra width that may be added to some synthesized fonts. When synthesizing some attributes, such as bold or italic, GDI or a device may have to add width to a string on both a per character and per string basis. For example, GDI emboldens a string by expanding the intracharacter spacing and overstriking with an offset, and italicizes a font by skewing the string. In either case there is an overhang past the basic string. For bold strings, it is the distance by which the overstrike is offset. For italic strings, it is the amount the top of the font is skewed past the bottom of the font. <i>tmOverhang</i> allows the application to determine how much of the character width returned by a GetTextExtent call on a single character is the actual character width and how much is the per-string extra width. The actual width is the extent minus the overhang.
tmDigitizedAspectX, tmDigitizedAspectY	Specify the aspect ratio of the device for which this font was designed. The ratio of <i>tmDigitizedAspectY</i> to <i>tmDigitizedAspectX</i> can be compared to the ratio of AspectY to AspectX retrieved from GetDeviceCaps .

Drawing Modes

Drawing modes define how GDI combines source and destination colors when drawing with the current pen. The drawing modes are actually binary raster operation codes, representing all possible Boolean functions of two variables, using the binary operations AND, OR, and XOR (exclusive OR) and the unary operation NOT. The table below lists the drawing modes and the Boolean operations they represent. Drawing modes are used in the **SetROP2** function.

Drawing Mode		Functions of Dest and Pen		
R2_BLACK	1	/*	O	*/
R2_NOTMERGESEN	2	/*	DPon	*/
R2_MASKNOTPEN	3	/*	DPna	*/
R2_NOTCOPYPEN	4	/*	Pn	*/
R2_MASKPENNOST	5	/*	PDna	*/
R2_NOT	6	/*	Dn	*/
R2_XORPEN	7	/*	DPx	*/
R2_NOTMASKPEN	8	/*	DPan	*/
R2_MASKPEN	9	/*	DPA	*/
R2_NOTXORPEN	10	/*	DPxn	*/
R2_NOP	11	/*	D	*/
R2_MERGENOTPEN	12	/*	DPno	*/
R2_COPYPEN	13	/*	P	*/
R2_MERGEPPENNOT	14	/*	PDno	*/
R2_MERGEPPEN	15	/*	DPO	*/
R2_WHITE	16	/*	1	*/

Raster Operation Codes

Raster operation codes (Rops) define how GDI combines colors in output operations that involve a current brush, a possible source bitmap, and a destination bitmap. The codes are actually tertiary raster operation codes, representing all possible Boolean functions of three variables, using the binary operations AND, OR, and XOR (exclusive OR) and the unary operation NOT.

The table below lists some of the codes and the Boolean operations they represent.

SRCCOPY	0x00CC0020	/*dest=source	*/
SRCPAINT	0x00EE0086	/*dest=source OR dest	*/
SRCAND	0x008800C6	/*dest=source AND dest	*/
SRCINVERT	0x00660046	/*dest= source XOR dest	*/
SRCERASE	0x00440328	/*dest= source AND (not dest)	*/
NOTSRCCOPY	0x00330008	/*dest= (not source)	*/
NOTSRCERASE	0x001100A6	/*dest= (not source) AND (not dest)	*/
MERGECOPY	0x00C000CA	/*dest= (source AND pattern)	*/
MERGEPAINT	0x00BB0226	/*dest= (source AND pattern) OR dest	*/
PATCOPY	0x00F00021	/*dest= pattern	*/
PATPAINT	0x00FBOA09	/*DPSnoo	*/
PATINVERT	0x005A0049	/*dest= pattern XOR dest	*/
DSTINVERT	0x00550009	/*dest= (not dest)	*/
BLACKNESS	0x00000042	/*dest= BLACK	*/
WHITENESS	0x00FF0062	/*dest= WHITE	*/

A complete list of the functions is given in Appendix A.

METAFILEPICT - *Metafile Picture Structure*

The **METAFILEPICT** structure defines the metafile picture format used for exchanging metafile data through the clipboard. The structure contains the following fields:

```
int      mm
int      xExt
int      yExt
HANDLE   hMF
```

The fields of the **METAFILEPICT** structure have the following meanings:

Field	Definition
mm	Gives the mapping mode in which the picture is drawn.
xExt, yExt	Gives the size of the metafile picture for all modes except MM_ISOTROPIC and MM_ANISOTROPIC. The x and y extents specify the rectangle within which the picture is drawn. The coordinates are in units that correspond to the mapping mode.
	For MM_ISOTROPIC and MM_ANISOTROPIC modes, which are scalable, xExt and yExt contain an optional "suggested size" in MM_HIMETRIC units. For MM_ANISOTROPIC pictures, xExt and yExt can be 0 when no suggested size is supplied. For MM_ISOTROPIC pictures, an aspect ratio must be

supplied even when no suggested size is given. (If a suggested size is given, the aspect ratio is implied by the size.) To give an aspect ratio without implying a suggested size, set *xExt* and *yExt* to negative values whose ratio is the appropriate aspect ratio. The magnitude of the negative *xExt* and *yExt* values will be ignored; only the ratio will be used.

mHF Handle to a memory metafile.

5.5 Communication Data Structures

This section describes the data structures used with the communication functions.

DCB – *Communications Device Control Block*

The **DCB** structure defines the control setting for the serial communications device. The structure contains the following fields:

BYTE	Id
WORD	BaudRate
BYTE	ByteSize
BYTE	Parity
BYTE	StopBits
WORD	RtsTimeout
WORD	CtsTimeout
WORD	DsrTimeout
BYTE	fBinary: 1
BYTE	fRtsDisable: 1
BYTE	fParity: 1
BYTE	fOutxCtsFlow:1
BYTE	fOutxDsrFlow:1
BYTE	fDummy: 2
BYTE	fDtrDisable:1

BYTE	fOutX: 1
BYTE	fInX: 1
BYTE	fPeChar: 1
BYTE	fNull: 1
BYTE	fChEvt: 1
BYTE	fDtrflow: 1
BYTE	fRtsflow: 1
BYTE	fDummy2: 1
char	XonChar
char	XoffChar
WORD	XonLim
WORD	XoffLim
char	PeChar
char	EofChar
char	EvtChar
WORD	TxDelay

The **DCB** fields have the following meanings:

Field	Definition
Id	Identifies the communication device. This value is set by the device driver. If the most significant bit is set, then the DCB is for a parallel device.
BaudRate	Specifies the baud rate at which the communication device is to operate..
ByteSize	Specifies the number of bits in the characters to be transmitted and received. It can be any number in the range 4 to 8.
Parity	Specifies the parity scheme to be used. It can be any one of the following: NOPARITY - No parity ODDPARITY - Odd EVENPARITY - Even MARKPARITY - Mark SPACEPARITY - Space
StopBits	Specifies the number of stop bits to be used. It can be any one of the following: ONESTOPBIT - 1 stop bit ONE5STOPBITS - 1.5 stop bits TWOSTOPBITS - 2 stop bits

RlsTimeout	Specifies the maximum amount of time, in milliseconds, the device should wait for the Receive Line Signal Detect signal (RLSD) to become high. (RLSD is also known as the Carrier Detect (CD) signal.)
CtsTimeout	Specifies the maximum amount of time, in milliseconds, the device should wait for the Clear To Send signal (CTS) to become high.
DsrTimeout	Specifies the maximum amount of time, in milliseconds, the device should wait for the Data Set Ready signal (DSR) to become high.
fBinary	Indicates binary mode. In nonbinary mode, <i>EofChar</i> is recognized on input and remembered as the end of data.
fRtsDisable	Indicates whether or not the Request To Send signal (RTS) is disabled. If set, RTS is not used and remains low. If clear, RTS is asserted when the device is opened, and dropped when closed.
fParity	Indicates whether or not parity checking is enabled. If set, parity checking is performed and errors are reported.
fOutxCtsFlow	Indicates that Clear To Send (CTS) is to be monitored for output flow control. If set and CTS drops, output is suspended until CTS is again asserted.
fOutxDsrFlow	Indicates that Data Set Ready (DSR) is to be monitored for output flow control. If set and DSR drops, output is suspended until DSR is again asserted.
fDtrDisable	Indicates whether or not the Data Terminal Ready signal (DTR) is disabled. If set, DTR is not used and remains low. If clear, DTR is asserted when the device is opened, and dropped when closed.
fOutX	Indicates that XON/XOFF flow control is to be used during transmission. If set, transmission stops when the <i>XoffChar</i> character is received, and starts again when the <i>XonChar</i> character is received.
fInX	Indicates that XON/XOFF flow control is to be used during reception. If set, the <i>XonChar</i> character is sent when the receive queue comes within <i>XoffLim</i> characters of being full, and the <i>XonChar</i> character is sent when the receive queue comes within <i>XonLim</i> characters of being empty.

fPeChar	Indicates that characters received with parity errors are to be replaced with the character specified by <i>PeChar</i> . <i>fParity</i> must be set for the replacement to occur.
fNull	Indicates that received NULL characters are to be discarded.
fChEvt	Indicates that reception of the <i>EvtChar</i> is to be flagged as an event.
fDtrFlow	Indicates that the Data Terminal Ready (DTR) signal is to be used for receive flow control. If set, DTR is dropped when the receive queue comes within <i>XoffLim</i> characters of being full, and asserted when the receive queue comes within <i>XonLim</i> characters of being empty.
fRtsFlow	Indicates that the Ready To Send (RTS) signal is to be used for receive flow control. If set, RTS is dropped when the receive queue comes within <i>XoffLim</i> characters of being full, and asserted when the receive queue comes within <i>XonLim</i> characters of being empty.
XonChar	Specifies the ASCII value of the XON character for both transmission and reception.
XoffChar	Specifies the ASCII value of the XOFF character for both transmission and reception.
XonLim	Specifies the minimum number of characters allowed in the receive queue before the XON character is sent.
XoffLim	Specifies the margin for the maximum number of characters allowed in the receive queue before the XOFF character is sent. The <i>XoffLim</i> value is subtracted from the size of the receive queue (in bytes) to calculate the maximum number of characters allowed.
PeChar	Specifies the ASCII value of the character used to replace characters received with a parity error.

EvtChar	Specifies the ASCII value of the character used to signal an event.
EofChar	Specifies the ASCII value of the character used to signal the end of data.
TxDelay	Specifies the minimum amount of time that must pass between transmission of characters.

COMSTAT – Communication Device Status

The **COMSTAT** structure contains information about a communications device. The structure contains the following fields:

```
BYTE    fCtsHold: 1
BYTE    fDsrHold: 1
BYTE    fRlsdHold: 1
BYTE    fXoffHold: 1
BYTE    fXoffSent: 1
BYTE    fEof: 1
BYTE    fTxim: 1
WORD   cbInQue
WORD   cbOutQue
```

The **COMSTAT** fields have the following meanings:

Field	Definition
fCtsHold	Indicates whether or not transmission is waiting for the Clear To Send (CTS) signal to be asserted.
fDsrHold	Indicates whether or not transmission is waiting for the Data Set Ready (DSR) signal to be asserted.
fRlsdHold	Indicates whether or not transmission is waiting for the Receive Line Signal Detect (RLSD) signal to be asserted.
fXoffHold	Indicates whether or not transmission is waiting as a result of the <i>XoffChar</i> being received.
fXoffSent	Indicates whether or not transmission is waiting as a result of the <i>XoffChar</i> being transmitted. Transmission halts when the <i>XoffChar</i> is transmitted to facilitate systems that take the next character as XON, regardless of the actual character.

fEof	Indicates whether or not the <i>EofChar</i> has been received.
fTxim	Indicates whether or not a character is waiting to be transmitted "immediately."
cbInQue	Specifies the number of characters in the receive queue.
cbOutQue	Specifies the number of characters in the transmit queue.

5.6 Open File Structure

OFSTRUCT – *Open File Structure*

The **OFSTRUCT** structure contains the following fields:

BYTE	cBytes
BYTE	fFixedDisk
WORD	nErrCode
BYTE	reserved[4]
BYTE	szPathName[128]

The fields have the following meaning:

Field	Definition
cBytes	Length of OFSTRUCT structure in bytes.
fFixedDisk	One byte specifying whether or not the file is on a fixed disk. It is nonzero if located on a fixed disk.
nErrCode	An unsigned short integer specifying the DOS error code if OpenFile failed (i.e., returns -1).
reserved	Four bytes reserved for future use.
szPathName	128 bytes containing the pathname of the file.



Chapter 6

File Formats

6.1	Introduction	281
6.2	Resource Script File	281
6.2.1	Single Line Statements	282
6.2.2	User-Defined Resources	283
6.2.3	STRINGTABLE Statement	284
6.2.4	ACCELERATORS Statement	286
6.2.5	MENU Statement	287
6.2.6	DIALOG Statement	292
6.2.7	Directives	307
6.3	Module Definition File	311
6.4	Windows Initialization File	318
6.4.1	Windows Section	320
6.4.2	Ports Section	321
6.4.3	Colors Section	321
6.4.4	Devices Section	322
6.4.5	Fonts Section	323
6.4.6	International Section	323

1

2

3

6.1 Introduction

This chapter describes the file formats of the ASCII text files used to create and execute a Windows application. There are the following file types:

- Resource Script File
- Module Definition File
- Windows Initialization File

The following sections describe each type in detail.

6.2 Resource Script File

The resource script file defines the names and attributes of the resources to be added to the application's executable file. The file consists of one or more "resource statements" that define the resource type and original file. The following is a list of the resource statements:

Single-line statements	CURSOR ICON BITMAP FONT
User-defined resources	
Multiple-line statements	STRINGTABLE ACCELERATORS MENU DIALOG
Directives	#include #define #undef #ifdef #ifndef #if #elif #else #endif

The following sections describe these statements in detail.

6.2.1 Single Line Statements

The single line statements define resources that are contained in a single file, such as cursors, icons, and fonts. The statements associate the filename of the resource with an identifying name or number. The resource is added to the executable file when the application is created, and can be extracted during execution by referring to the name or number.

The general form for all single line statements is:

nameID *resource-type* [*load-option*] [*mem-option*] *filename*

nameID is either a unique name or an integer number identifying the resource. For a **FONT** resource, the *nameID* must be a number; it cannot be a name.

resource-type is one of the following keywords, specifying the type of resource to be loaded:

Keyword	Resource Type
CURSOR	A cursor resource is a bitmap defining the shape of the mouse cursor on the display screen.
ICON	An icon resource is a bitmap defining the shape of the icon to be used for a given application.
BITMAP	A bitmap resource is a custom bitmap that an application intends to use in its screen display or as an item in a menu.
FONT	A font resource is simply a file containing a font. The format of a font file is defined in Appendix C.

load-option is an optional keyword specifying when the resource is to be loaded. It must be one of the following:

PRELOAD	Resource is loaded immediately
LOADONCALL	Resource is loaded when called

The default is LOADONCALL.

The *mem-option* consists of the following keyword or keywords, specifying whether the resource is fixed or moveable and whether it is discardable:

FIXED	Resource remains at a fixed memory location
MOVEABLE	Resource can be moved if necessary to compact memory
DISCARDABLE	Resource can be discarded if no longer needed

The default is MOVEABLE and DISCARDABLE for **CURSOR**, **ICON**, and **FONT** resources. The default for **BITMAP** resources is MOVEABLE.

filename is an ASCII string specifying the MS-DOS filename of the file containing the resource. A full pathname must be given if the file is not in the current working directory.

Examples

```

pointer CURSOR point.cur
pointer CURSOR DISCARDABLE point.cur
10      CURSOR custom.cur

desk     ICON desk.ico
desk     ICON DISCARDABLE desk.ico
11      ICON custom.ico

disk     BITMAP disk.bmp
disk     BITMAP DISCARDABLE disk.bmp
12      BITMAP custom.bmp

5 FONT   CMROMAN.FON

```

6.2.2 User-Defined Resources

An application can also define its own resource. The resource can be any data that the application intends to use. A user-defined resource statement has the form:

nameID *typeID* [*load-option*] [*mem-option*] *filename*

nameID is either a unique name or an integer number identifying the resource.

typeID is either a unique name or an integer number identifying the resource type. If a number is given, it must be greater than 255. The type numbers 1 through 255 are reserved for existing and future predefined resource types.

load-option is an optional keyword specifying when the resource is to be loaded. It must be one of the following:

PRELOAD	Resource is loaded immediately
LOADONCALL	Resource is loaded when called

The default is LOADONCALL.

mem-option consists of the following keyword or keywords, specifying whether the resource is fixed or moveable and whether it is discardable:

FIXED	Resource remains at a fixed memory location
MOVEABLE	Resource can be moved if necessary to compact memory
DISCARDABLE	Resource can be discarded if no longer needed

The default is MOVEABLE.

filename is an ASCII string specifying the MS-DOS filename of the file containing the cursor bitmap. A full pathname must be given if the file is not in the current working directory.

Example

```
array    MYRES   data.res
14      300     custom.res
```

6.2.3 STRINGTABLE Statement

The **STRINGTABLE** statement defines one or more string resources for an application. String resources are simply null-terminated ASCII strings that can be loaded when needed from the executable file, using the **LoadString** function.

The **STRINGTABLE** statement has the form:

```
 STRINGTABLE [load-option] [mem-option]  
 BEGIN  
   string-definitions  
 END
```

where *string-definitions* are one or more ASCII strings, enclosed in double quotation marks and preceded by an identifier. The identifier must be an integer.

load-option is an optional keyword specifying when the resource is to be loaded. It must be one of the following:

PRELOAD	Resource is loaded immediately
LOADONCALL	Resource is loaded when called

The default is LOADONCALL.

The optional *mem-option* consists of the following keyword or keywords, specifying whether the resource is fixed or moveable and whether it is discardable:

FIXED	Resource remains at a fixed memory location
MOVEABLE	Resource can be moved if necessary to compact memory
DISCARDABLE	Resource can be discarded if no longer needed

The default is MOVEABLE and DISCARDABLE.

Example

```
#define IDS_HELLO      1  
#define IDS_GOODBYE    2  
  
 STRINGTABLE  
 BEGIN  
   IDS_HELLO,      "Hello"  
   IDS_GOODBYE,    "Goodbye"  
 END
```

6.2.4 ACCELERATORS Statement

The **ACCELERATORS** statement defines one or more accelerators for an application. An accelerator is a keystroke defined by the application to give the user a quick way to perform a task. The **TranslateAccelerator** function is used to translate accelerator messages from the application queue into WM_COMMAND or WM_SYSCOMMAND messages.

The **ACCELERATORS** statement has the form:

```
acctablename ACCELERATORS
BEGIN
event, idvalue [ , type ] [, NOINVERT] [, SHIFT] [, CONTROL]
.
.
.
END
```

acctablename is the name of the accelerator table.

event is the keystroke to be used as an accelerator. It can be any one of the following:

Character	Meaning
“char”	A single character enclosed in double quotes. The character can be preceded by a caret (^), meaning that the character is a control character.
ASCII character	An integer value representing an ASCII character. The <i>type</i> must be ASCII.
Virtual key character	An integer value representing a virtual key. The <i>type</i> field must be VIRTKEY.

idvalue is an integer value identifying the accelerator.

The *type* field is required only when *event* is an ASCII character or virtual key character. *type* is either ASCII or VIRTKEY; the integer value of *event* is interpreted accordingly.

The NOINVERT flag, if given, means that no top-level menu item is highlighted when the accelerator is used. This is useful, for example, when defining accelerators for actions such as scrolling that do not correspond to a menu item. If NOINVERT is omitted, a top-level menu item will be highlighted (if possible) when the accelerator is used.

The SHIFT flag, if given, causes the accelerator to be activated only if the shift key is down.

The CONTROL flag, if given, defines the character as a control character (the accelerator is only activated if the control key is down). This has the same effect as using a caret (^) before the accelerator character in the *event* field.

Examples

```
MainAcc ACCELERATORS
BEGIN
    "^\$", ID_SAVE, NOINVERT
    VK_UP, 6, VIRTKEY, NOINVERT
    7, ID_BELL, ASCII
    "^\g", ID_BELL
    "G", ID_BELL, CONTROL
END
```

Note that the last three definitions are equivalent.

6.2.5 MENU Statement

The MENU statement defines the contents of a menu resource. A menu resource is a collection of information that defines the appearance and function of an application menu. A menu is a special input tool that lets a user select commands from a list of command names.

The MENU statement has the form:

```
menuID MENU [load-option] [mem-option]
BEGIN
item-definitions
END
```

menuID is a name or number used to identify the menu resource.

load-option is an optional keyword specifying when the resource is to be loaded. It must be one of the following:

PRELOAD	Resource is loaded immediately
LOADONCALL	Resource is loaded when called

The default is LOADONCALL.

The optional *mem-option* consists of the following keyword or keywords, specifying whether the resource is fixed or moveable and whether it is discardable:

FIXED	Resource remains at a fixed memory location
MOVEABLE	Resource can be moved if necessary to compact memory
DISCARDABLE	Resource can be discarded if no longer needed

item-definitions are special resource statements that define the items in the menu.

Example

The following is an example of a complete MENU statement.

```
sample MENU
BEGIN
    MENUITEM "Alpha", 100
    POPUP "Beta"
    BEGIN
        MENUITEM "Item 1", 200
        MENUITEM "Item 2", 201, CHECKED
    END
END
```

Item Definition Statements

The MENUITEM and POPUP statements are used in the *item-definition* section of a MENU statement to define the names and attributes of the actual menu items. Any number of statements can be given; each defines a unique item. The order of the statements define the order of the menu items.

Note

The MENUITEM and POPUP statements can only be used within an *item-definition* section of a MENU statement.

MENUITEM *text, result, optionlist*

Purpose This optional statement defines a menu item.

Parameters *text* is an ASCII string, enclosed in double quotation marks, specifying the name of the menu item. The string can contain the escape characters \t and \a. The \t character inserts a tab in the string when displayed and is used to align text in columns. Tab characters should be used only in popup menus, not in menu bars. The \a character right-justifies all text that follows it. To insert a double quote character ("") in the *text*, use two double quote characters ("").

result is an integer number specifying the result generated when the user selects the menu item. Menu item results are always integers and are sent to the window owning the menu when the user clicks on the menu item name.

optionlist is one or more predefined menu options, separated by commas or spaces, that specify the appearance of the menu item. The menu options are as follows:

Option	Meaning
MENUBREAK	Item is immediately preceded by a new line.
CHECKED	Item has a checkmark next to it.
INACTIVE	Item name is displayed, but cannot be selected.
GRAYED	Item name is initially inactive and drawn with a gray highlight.

The INACTIVE and GRAYED options cannot be used together.

Examples

```
MENUITEM "Alpha", 1, CHECKED, GRAYED
MENUITEM "Beta", 2
```

POPUP *text, optionlist*

Purpose This statement marks the beginning of a popup menu definition. A popup menu is a special menu item that displays a sublist of menu items when it is selected. The **POPUP** statement actually has the form:

```
POPUP text, optionlist
BEGIN
item-definitions
END
```

where *item-definitions* can be any number of **MENUITEM** statements. **POPUP** statements in a popup menu are not allowed.

Parameters *text* is an ASCII string, enclosed in double quotation marks, specifying the name of the popup.

optionlist is one or more predefined menu options that specify the appearance of the menu item. It can be any one of the following:

Option	Meaning
MENUBREAK	Item is placed in a new column.
MENUBARBREAK	Item is placed in a new column. The old and new columns are separated with a bar.
CHECKED	Item has a checkmark next to it.
INACTIVE	Item name is displayed, but cannot be selected.
GRAYED	Item name is initially inactive and drawn with a gray highlight.

The options can be combined using the bitwise OR operator. The **INACTIVE** and **GRAYED** options cannot be used together.

Example

```
chem MENU
BEGIN

POPUP "Elements"
BEGIN
    MENUITEM "Oxygen", 200
    MENUITEM "Carbon", 201, CHECKED
    MENUITEM "Hydrogen", 202
END

POPUP "Compounds", CHECKED
BEGIN
    MENUITEM "Glucose", 301
    MENUITEM "Sucrose", 302, CHECKED
    MENUITEM "Lactose", 303, MENUBREAK
    MENUITEM "Fructose", 304
END

END
```

MENUITEM SEPARATOR

Purpose This special form of the MENUITEM statement creates an inactive menu item that serves as a dividing bar between two active menu items. In popup menus, the bar is horizontal. In a regular menu, the bar is vertical.

Parameters None.

Example

```
MENUITEM "Roman", 206
MENUITEM SEPARATOR
MENUITEM "20 Point", 301
```

6.2.6 DIALOG Statement

The **DIALOG** statement defines a template that can be used by an application to create dialog boxes.

The dialog statement has the form:

```
dialogname DIALOG [load-option] [mem-option] x, y, width, height
option-statements
BEGIN
control-statements
END
```

The parts of the DIALOG statement are described below.

The name **DIALOG** can also be used as the class name parameter to the **CreateWindow** function to create a window with dialog box attributes.

nameID DIALOG [*load-option*] [*mem-option*] *x, y, width, height*

Purpose This statement marks the beginning of a **DIALOG** statement. It defines the name of the dialog box, memory and load options, the box's starting location on the display screen, its width, and its height.

Parameters *nameID* is either a unique name or an integer number identifying the resource.

load-option is an optional keyword specifying when the resource is to be loaded. It must be one of the following:

PRELOAD	Resource is loaded immediately
LOADONCALL	Resource is loaded when called

The default is LOADONCALL.

The optional *mem-option* consists of the following keyword or keywords, specifying whether the resource is fixed or moveable and whether it is discardable:

FIXED	Resource remains at a fixed memory location
MOVEABLE	Resource can be moved if necessary to compact memory
DISCARDABLE	Resource can be discarded if no longer needed

The default is MOVEABLE.

x and *y* are integer numbers specifying the *x* and *y* coordinates on the display screen of the upper left corner of the dialog box. The exact meaning of the coordinates depends on the style defined by the **STYLE** statement. For child style dialog boxes, the coordinates are relative to the origin of the parent window. For popup style boxes, the coordinates are relative to the origin of the display screen.

width and *height* are integer numbers specifying the width and height of the box. The width units are 1/4 the width of a character; the height units are 1/8 the height of a character.

Example

```
errmess DIALOG 10, 10, 300, 200
```

The following is a complete example of a **DIALOG** statement.

```
#include "windows.h"

errmess DIALOG 10, 10, 300, 110
STYLE WS_POPUP|WS_BORDER
CAPTION "Error!"
BEGIN
    CTEXT "Select One:", 1, 10, 10, 280, 12
    RADIOBUTTON "Retry", 2, 75, 30, 60, 12
    RADIOBUTTON "Abort", 3, 75, 50, 60, 12
    RADIOBUTTON "Ignore", 4, 75, 80, 60, 12
END
```

Dialog Option Statements

The dialog option statements define special attributes of the dialog box, such as its style, caption, and menu. The option statements are optional. If not given, the dialog box is given a default attribute.

STYLE *style*

<i>Purpose</i>	This optional statement defines the window style of the dialog box. The window style specifies whether the box is a popup or a child window.
----------------	--

The default style has the following attributes:

WS_POPUP
WS_BORDER
WS_SYSMENU

Parameters *style* is an integer value or predefined name specifying the window style. It can be any of the window styles defined for the **CreateWindow** function (see Chapter 2, "Window Functions").

The style DS_SYSMODAL can also be used to create a system modal dialog box.

Notes If the predefined names are used, the #include directive must be used to include the "WINDOWS.H" file in the resource script.

CAPTION *captiontext*

Purpose This optional statement defines the dialog box's title. The title appears in the box's caption bar (if it has one).

The default caption is empty.

Parameters *captiontext* is a ASCII character string enclosed in double quotation marks.

Example

```
CAPTION "Error!"
```

MENU *menuname*

Purpose This optional statement defines the dialog box's menu.

The default menu is empty.

Parameters *menuname* is the resource name or number of the menu to be used.

Example

```
MENU errmenu
```

CLASS *class*

Purpose This optional statement defines the class of the dialog box.

Parameter *class* is an integer or a string (enclosed in double quotes) that identifies the dialog box class.

Example

```
CLASS "myclass"
```

Control Statements

The control statements, given in the *control-definition* section of the **DIALOG** statement, define the attributes of the control windows that appear in the dialog box. A dialog box is empty unless one or more control statements are given.

Control statements have the following general form:

```
control-type text, id, x, y, width, height[, style]
```

Three control statements are exceptions to this general form: the **EDIT-TEXT** and **LISTBOX** controls do not have a *text* field, and the custom control type, **CONTROL**, has one additional field, *class*.

The *control-type* field is one of the keywords described below, defining the type of the control.

text is an ASCII string specifying the text to be displayed. The string must be enclosed in double quotation marks. The manner in which the text is displayed depends on the particular control, as detailed below.

id is a unique integer number identifying the control.

x and *y* are integer numbers specifying the *x* and *y* coordinates of the upper left corner of the control. The coordinates are relative to the origin of the dialog box.

width and *height* are integer numbers specifying the width and height of the control. The width units are 1/4 the width of a character; the height units are 1/8 the height of a character.

The *x*, *y*, *width*, and *height* fields can use addition and subtraction operators (+ and -) for relative positioning. For example, "15 + 6" can be used for the *x* field.

The optional *style* field consists of one or more of the control styles given later in this chapter in Table 6.2 and the window styles defined in Chapter 2. Styles can be combined using the bitwise OR operator.

The *control-type* keywords are described below, and their class and default style are given. See Tables 6.1 and 6.2 for a full description of control classes and styles.

LTEXT

Description Left-justified text control. A simple rectangle displaying the given *text* left-justified in the rectangle. The *text* is formatted before it is displayed. Words that would extend past the end of a line are automatically wrapped to the beginning of the next line.

Class Static

Default Style ES_LEFT, WS_GROUP

RTEXT

Description Right-justified text control. A simple rectangle displaying the given *text* right-justified in the rectangle. The *text* is formatted before it is displayed. Words that would extend past the end of a line are automatically wrapped to the beginning of the next line.

Class Static

Default Style ES_RIGHT, WS_GROUP

CTEXT

Description Centered text control. A simple rectangle displaying the given *text* centered in the rectangle. The *text* is formatted before it is displayed. Words that would extend past the end of a line are automatically wrapped to the beginning of the next line.

Class Static

Default Style ES_CENTER, WS_GROUP

CHECKBOX

- Description* A small rectangle (check box) that is highlighted when clicked. The given *text* is displayed just to the right of the check box. The control highlights the square when the user clicks the mouse in it, and removes the highlight on the next click.
- Class* Button
- Default Style* BS_CHECKBOX, WS_TABSTOP

PUSHBUTTON

- Description* A rectangle containing the given *text*. The control sends a message to its parent whenever the user clicks the mouse inside the rectangle.
- Class* Button
- Default Style* BS_PUSHBUTTON, WS_TABSTOP

LISTBOX

- Description* A rectangle containing a list of strings (such as filenames) from which the user can make selections. The LISTBOX control statement does not contain a *text* field, so the form of the LISTBOX statement is:

LISTBOX *id, x, y, cx, cy [, style]*

The fields have the same meaning as in the other control statements.

- Class* List box
- Default Style* LBS_NOTIFY, LBS_SORT, WS_VSCROLL, WS_BORDER

GROUPBOX

- Description* A rectangle that groups other controls together. The controls are grouped by drawing a border around them and displaying the given *text* in the upper left corner.
- Class* Button
- Default Style* BS_GROUPBOX, WS_TABSTOP

DEFPUSHBUTTON

Description A small rectangle with an emboldened outline that represents the default response for the user. The *text* is displayed inside the button. The control highlights the button in the usual way when the user clicks the mouse in it and sends a message to its parent window.

Class Button

Default Style BS_DEFPUSHBUTTON, WS_TABSTOP

RADIOBUTTON

Description A small rectangle that has the given *text* displayed just to its right. The control highlights the square when the user clicks the mouse in it and sends a message to its parent window. The control removes the highlight and sends a message on the next click.

Class Button

Default Style BS_RADIOBUTTON, WS_TABSTOP

EDITTEXT

Description A rectangle in which the user can enter and edit text. The control displays a cursor when the user clicks the mouse in it. The user can then use the keyboard to enter text or edit the existing text. Editing keys include the backspace and delete keys. The mouse can be used to select the character or characters to be deleted, or select the place to insert new characters.

The **EDITTEXT** control statement does not contain a *text* field, so its form is:

EDITTEXT *id, x, y, width, height [, style]*

The fields have the same meaning as in the other control statements.

Class Edit

Default Style WS_TABSTOP, ES_LEFT, WS_BORDER

ICON

Description An icon displayed in the dialog box. The given *text* is the name of an icon (not a filename) defined elsewhere in the resource file.

For the **ICON** statement, the *width* and *height* parameters are ignored; the icon automatically sizes itself.

Class Static

Default Style SS_ICON

CONTROL

Description A user-defined control window. In addition to the standard control statement fields, the **CONTROL** statement contains *class* and *style* fields, as described below. The form of the statement is:

CONTROL *text, id, class, style, x, y, width, height*

class is a predefined name, character string, or integer defining the the class. It can be any one of the following:

BUTTON
STATIC
EDIT
"*classname*"
integer

classname is an ASCII character string identifying a class.

integer is an integer value identifying a class.

style is a predefined name or integer number specifying the style of the given control. The exact meaning of *style* depends on the *class* value. Table 6.2 lists the control classes and corresponding styles.

Class Depends on *class* field.

Default Style None.

Table 6.1
Control Classes

Class	Meaning
BUTTON	A button control is a small rectangular child window that represents a "button" that the user can turn on or off by clicking on it with the mouse. Button controls can be used alone or in groups, and can either be labelled or appear without text. Button controls typically change appearance when the user clicks on them.
EDIT	An edit control is a rectangular child window in which the user can enter text from the keyboard. The user selects the control, and gives it the input focus, by clicking the mouse inside it or tabbing to it. The user can enter text when the control displays a flashing caret. The mouse can be used to move the cursor and select characters to be replaced, or position the cursor for inserting characters. The backspace key can be used to delete characters.
	Edit controls use the fixed-pitch font and display ANSI characters. They expand tab characters into as many space characters as required to move the cursor to the next tab stop. Tab stops are assumed to be at every eighth character position.
STATIC	Static controls are simple text fields, boxes, and rectangles that can be used to label, box, or separate other controls. Static controls take no input and provide no output.
LISTBOX	List box controls consist of a list of character strings. The control is used whenever an application needs to present a list of names, such as filenames, that the user can view and select. The user can select a string by pointing the mouse to the string and clicking a mouse button. Selected strings are highlighted and a notification message is passed to the parent window. A scroll bar can be used with a list box control to scroll lists too long or wide for the control window.

Table 6.1 (continued)

Class	Meaning
SCROLLBAR	A scroll bar control is a rectangle containing a thumb and direction arrows at both ends. The scrolling bar sends a notification message to its parent whenever the user clicks the mouse in the control. The parent is responsible for updating the thumb position, if necessary. Scroll bar controls have the same appearance and function as the scroll bars used in ordinary windows. Unlike scroll bars, scroll bar controls can be positioned anywhere in a window and used whenever needed to provide scrolling input for a window. The scroll bar class also includes size box controls. A size box control is a small rectangle that the user can expand to change the size of the window.
<i>Note</i>	A control class name can be used as the class name parameter to the CreateWindow function to create a child window having the control class attributes.

Table 6.2
Control Styles

Style	Meaning
BUTTON Class	
BS_PUSHBUTTON	Same as PUSHBUTTON statement.
BS_DEFPUSHBUTTON	Same as DEFPUSHBUTTON statement.
BS_CHECKBOX	Same as CHECKBOX statement.
BS_AUTOCHECKBOX	Button automatically toggles its state whenever the user clicks on it.
BS_RADIOBUTTON	Same as RADIobutton statement.
BS_3STATE	Identical to BS_CHECKBOX except that a button can be grayed as well as checked or unchecked. The grayed state is typically used to show that a check box has been disabled.
BS_AUTO3STATE	Identical to BS_3STATE except that the button automatically toggles its state when the user clicks on it.
BS_GROUPBOX	Same as GROUPBOX statement.
BS_USERBUTTON	User-defined button. Parent is notified when the button is clicked. Notification includes a request to paint, invert, and disable the button when necessary.
EDIT Class	
ES_LEFT	Left-justified text.
ES_CENTER	Centered text.
ES_RIGHT	Right-justified text.

Table 6.2 (*continued*)

Style	Meaning
ES_MULTILINE	<p>Multiple line edit control. (The default is single line.) If the AUTO_VSCROLL style is specified, the edit control shows as many lines as possible and scrolls vertically when a carriage return/line feed combination (CR/LF) is typed (but not when just a line feed is typed). If AUTO_VSCROLL is not given, the edit control shows as many lines as possible and beeps if the user enters CR/LF when no more lines can be displayed.</p> <p>If the ES_AUTOHSCROLL style is specified, the multiple line edit control automatically scrolls horizontally when the caret goes past the right edge of the control. To start a new line, the user must enter a carriage return/line feed combination. If ES_AUTOHSCROLL is not given, the control automatically word wraps when necessary; the user can also start a new line by entering CR/LF. The position of the word wrap is determined by the window size. If the window size changes, the word wrap position changes, and the text is redisplayed.</p>
ES_AUTOVSCROLL	Multiple line edit controls can have scroll bars. An edit control with scroll bars processes its own scroll bar messages. Edit controls without scroll bars scroll as described above, and process any scroll messages sent by the parent window.
ES_AUTOHSCROLL	Text is automatically scrolled up one page vertically when the user presses the carriage return on the last line.
ES_NOHIDESEL	Text is automatically scrolled horizontally to the right by 10 characters when the user types a character at the end of the line. When the user presses the carriage return, the control scrolls all text back to position 0.
	Normally, an edit control hides the selection when it loses the input focus, and inverts the selection when it receives the input focus. Specifying ES_NOHIDESEL overrides this default action.

Table 6.2 (*continued*)

Style	Meaning
<hr/>	
STATIC Class	
SS_LEFT	Same as LTEXT control
SS_CENTER	Same as CTEXT control
SS_RIGHT	Same as RTEXT control
SS_ICON	Same as ICON control
SS_BLACKRECT	Black filled rectangle
SS_GRAYRECT	Gray filled rectangle
SS_WHITERECT	White filled rectangle
SS_BLACKFRAME	Box with black frame
SS_GRAYFRAME	Box with gray frame
SS_WHITEFRAME	Box with white frame
SS_USERITEM	User-defined item
<hr/>	
LISTBOX Class	
LBS_NOTIFY	The parent receives an input message whenever the user clicks or double clicks a string.
LBS_MULTIPLESEL	The string selection is toggled each time the user clicks or double clicks on the string. Any number of strings can be selected.
LBS_SORT	The strings in the list box are sorted alphabetically.
LBS_NOREDRAW	The list box display is not updated when changes are made. This style can be changed at any time by sending a WM_SETREDRAW message.

Table 6.2 (*continued*)

Style	Meaning
SCROLLBAR Class	
SBS_VERT	Vertical scroll bar. If neither SBS_RIGHTALIGN nor SBS_LEFTALIGN is specified, the scroll bar has the height, width, and position given in the control statement or the CreateWindow call.
SBS_RIGHTALIGN	Used with SBS_VERT. The right edge of the scroll bar is aligned with the right edge of the rectangle specified by the <i>x</i> , <i>y</i> , <i>width</i> , and <i>height</i> values given in the control statement (or in the CreateWindow call). The scroll bar has the default width for system scroll bars.
SBS_LEFTALIGN	Used with SBS_VERT. The left edge of the scroll bar is aligned with the left edge of the rectangle specified by the <i>x</i> , <i>y</i> , <i>width</i> , and <i>height</i> values given in the control statement (or in the CreateWindow call). The scroll bar has the default width for system scroll bars.
SBS_HORZ	Horizontal scroll bar. If neither SBS_BOTTOMALIGN nor SBS_TOPALIGN is specified, the scroll bar has the height, width, and position given in the control statement or the CreateWindow call.
SBS_TOPALIGN	Used with SBS_HORZ. The top edge of the scroll bar is aligned with the top edge of the rectangle specified by the <i>x</i> , <i>y</i> , <i>width</i> , and <i>height</i> values given in the control statement (or in the CreateWindow call). The scroll bar has the default height for system scroll bars.
SBS_BOTTOMALIGN	Used with SBS_HORZ. The bottom edge of the scroll bar is aligned with the bottom edge of the rectangle specified by the <i>x</i> , <i>y</i> , <i>width</i> , and <i>height</i> values given in the control statement (or in the CreateWindow call). The scroll bar has the default height for system scroll bars.
SBS_SIZEBOX	Size box. If neither SBS_SIZEBOXBOTTOMRIGHTALIGN nor SBS_SIZEBOXTOPLEFTALIGN is specified, the size box has the height, width, and position given in the control statement or the CreateWindow call.

Table 6.2 (continued)

Style	Meaning
SBS_SIZEBOXTOPLEFTALIGN	Used with SBS_SIZEBOX. The top left corner of the size box is aligned with the top left corner of the rectangle specified by the <i>x</i> , <i>y</i> , <i>width</i> , and <i>height</i> values given in the control statement (or in the CreateWindow call). The size box has the default size for system size boxes.
SBS_SIZEBOXBOTTOMRIGHTALIGN	Used with SBS_SIZEBOX. The bottom right corner of the size box is aligned with the bottom right corner of the rectangle specified by the <i>x</i> , <i>y</i> , <i>width</i> , and <i>height</i> values given in the control statement (or in the CreateWindow call). The size box has the default size for system size boxes.
<hr/>	
All Classes	
WS_GROUP	Specifies the first control of a group of controls in which the user can move from one control to the next by using the cursor keys. All controls defined after the first control with WS_GROUP style belong to the same group. The next control with WS_GROUP style ends the first group and starts the next group (i.e., one group ends where the next begins).
WS_TABSTOP	Specifies one or any number of controls through which the user can move by tabbing. The TAB key moves the user to the next control with WS_TABSTOP style.

Examples

```
LTEXT "Enter Name:", 3, 10, 10, 40, 10
RTEXT "Number of Messages", 4, 30, 50, 100, 10
CTEXT "Title", 3, 10, 50, 40, 10
CHECKBOX "Arabic", 3, 10, 10, 40, 10
PUSHBUTTON "ON", 7, 10, 10, 20, 10
LISTBOX 666, 10, 10, 50, 54
GROUPBOX "Output", 42, 10, 10, 30, 50
RADIOBUTTON "AM 101", 10, 10, 10, 40, 10
EDITTEXT 3, 10, 10, 100, 10
```

6.2.7 Directives

The resource directives are special statements that define actions to perform on the script file before it is compiled. The directives can assign values to names, include the contents of files, and control compilation of the script file.

The resource directives are identical to the directives used in the C programming language. They are fully defined in the *Microsoft C Reference Manual*.

#include *filename*

Purpose This directive copies the contents of the file specified by *filename* into your resource script before **rc** processes the script.

Parameters *filename* is an ASCII string, enclosed in double quotation marks, specifying the MS-DOS filename of the file to be included. A full pathname must be given if the file is not in the current directory or in the directory specified by the **INCLUDE** environment variable.

The *filename* parameter is handled as a C string, and two backslashes must be given wherever one is expected in the pathname (for example, "root\\sub".) Or, a single forward slash (/) can be used instead of double backslashes (for example, "root/sub".)

Example

```
#include "windows.h"

PenSelect MENU
BEGIN
    MENUITEM "black pen", BLACK_PEN
END
```

#define name value

Purpose This directive assigns the given *value* to *name*. All subsequent occurrences of *name* are replaced by the *value*.

Parameters *name* is any combination of letters, digits, or punctuation.
value is any integer number, character string, or line of text.

Examples

```
#define nonzero 1  
#define USERCLASS "MyControlClass"
```

#undef name

Purpose This directive removes the current definition of *name*. All subsequent occurrences of *name* are processed without replacement.

Parameters *name* is any combination of letters, digits, or punctuation.

Examples

```
#undef nonzero  
#undef USERCLASS
```

#ifdef name

Purpose This directive carries out conditional compilation of the resource file by checking the specified *name*. If the *name* has been defined using a **#define** directive, **#ifdef** directs the resource compiler to continue with the statement immediately after it. If *name* has not been defined, **#ifdef** directs the compiler to skip all statements up to the next **#endif** directive.

Parameters *name* is the name to be checked by the directive.

Example

```
#ifdef Debug  
errbox BITMAP errbox.bmp  
#endif
```

#ifndef *name**Purpose*

This directive carries out conditional compilation of the resource file by checking the specified *name*. If the *name* has not been defined or if its definition has been removed using the #**undef** directive, #**ifndef** directs the resource compiler to continue processing statements up to the next #**endif**, #**else**, or #**elif** directive, then skip to the statement after after the #**endif**. If *name* is defined, #**ifndef** directs the compiler to skip to the next #**endif**, #**else**, or #**elif** directive.

Parameters

name is the name to be checked by the directive.

Example

```
#ifndef Optimize
errbox BITMAP errbox.bmp
#endif
```

#if *constant-expression**Purpose*

This directive carries out conditional compilation of the resource file by checking the specified *constant-expression*. If the *constant-expression* is nonzero #**if** directs the resource compiler to continue processing statements up to the next #**endif**, #**else**, or #**elif** directive, then skip to the statement after after the #**endif**. If *constant-expression* is zero, #**if** directs the compiler to skip to the next #**endif**, #**else**, or #**elif** directive.

Parameters

constant-expression is a defined name, an integer constant, or an expression consisting of names, integers, and arithmetic and relational operators.

Example

```
#if Version<3
errbox BITMAP errbox.bmp
#endif
```

#elif *constant-expression**Purpose*

This directive marks an optional clause of a conditional compilation block defined by an #**ifdef**, #**ifndef**, or #**if** directive. The directive carries out conditional compilation of the resource file by checking the specified *constant-expression*. If the *constant-expression* is nonzero #**elif** directs the resource compiler to continue processing statements up to the next #**endif**, #**else**, or #**elif** directive,

then skip to the statement after the **#endif**. If *constant-expression* is zero, **#elif** directs the compiler to skip to the next **#endif**, **#else**, or **#elif** directive. Any number of **#elif** directives can be used in a conditional block.

Parameters *constant-expression* is a defined name, an integer constant, or an expression consisting of names, integers, and arithmetic and relational operators.

Example

```
#if Version<3
errbox BITMAP errbox.bmp
#elif Version<7
errbox BITMAP userbox.bmp
#endif
```

#else

Purpose This directive marks an optional clause of a conditional compilation block defined by an **#ifdef**, **#ifndef**, or **#if** directive. The **#else** directive must be the last directive before **#endif**.

Parameters None.

Example

```
#ifdef Debug
errbox BITMAP errbox.bmp
#else
errbox BITMAP userbox.bmp
#endif
```

#endif

Purpose This directive marks the end of a conditional compilation block defined by an **#ifdef** directive. One **#endif** is required for each **#ifdef** directive.

Parameters None.

6.3 Module Definition File

The module definition file defines the contents and system requirements of a Windows application. The file contains one or more module statements, each defining a specific attribute of the application, such as its module name, the number and type of program segments, and the number and names of exported and imported functions.

There are the following module statements:

Statement	Description
NAME	Module name
LIBRARY	Library name
DESCRIPTION	One line description of the module
DATA	Data segment attributes
CODE	Code segment attributes
HEAPSIZE	Local heap size in bytes
STACKSIZE	Local stack size in bytes
SEGMENT	Additional code segment
EXPORTS	Exported functions
IMPORTS	Imported functions
STUB	Old style executable

The following are complete definitions of the module statements.

NAME *modulename*

Purpose This statement defines the name of the application's executable module. The name is used to identify the module when importing or exporting functions.

Parameters *modulename* is one or more ASCII characters.

Example

NAME Calendar

LIBRARY *libraryname*

- Purpose* This statement defines the name of a library module. Library modules are resource modules that contain code, data, and other resources but are not intended to be executed.
- Parameters* *libraryname* is one or more ASCII characters defining the name of the library module.
- Notes* The start address of the module determined by the object files. It is an internally defined function.
- Example*

LIBRARY User

DESCRIPTION *text*

- Purpose* This statement inserts *text* into the application's module. It is useful for embedding source control or copyright information.
- Parameters* *text* is one or more ASCII characters. The string must be enclosed in single quotation marks.
- Example*

DESCRIPTION 'Microsoft Windows Template Application'

HEAPSIZE *bytes*

- Purpose* This statement defines the number of bytes needed by the application for its local heap. An application uses the local heap whenever it allocates local memory.
- The default heapsize is 0.
- Parameters* *bytes* is an integer number specifying the heap size in bytes. It must not exceed 65,536 (the size of a single physical segment).
- Example*

HEAPSIZE 4096

STACKSIZE *bytes*

Purpose This statement defines the number of bytes needed by the application for its local stack. An application uses the local stack whenever it calls its own functions. A minimum stack size of 4096 bytes is recommended.

The default stack size is 0, if the application makes no function calls. Otherwise, it is 4096.

Parameters *bytes* is an integer number specifying the stack size in bytes.

Example

```
STACKSIZE 4096
```

CODE *memory-option load-option pure-option*

Purpose This statement defines the attributes of the application's code segment.

Parameters *memory-option* is an optional keyword specifying whether the segment is fixed or moveable. It must be one of the following:

FIXED Segment remains at a fixed memory location

MOVEABLE Segment can be moved if necessary to compact memory

DISCARDABLE Segment can be discarded if no longer needed

The default is FIXED.

load-option is an optional keyword specifying when the segment is to be loaded. It must be one of the following:

PRELOAD Segment is loaded immediately

LOADONCALL Segment is loaded when called

The default is PRELOAD.

pure-option is an optional keyword specifying whether the segment is pure or impure. It must be one of the following:

PURE Segment contains code only

IMPURE Segment contains a mixture of code and data

The default is PURE.

Notes PURE and IMPURE are used for 286 protected mode programs.

Example

```
CODE MOVEABLE LOADONCALL
```

DATA *memory-options instance-options*

Purpose This statement defines the attributes of the application's data segment. The data segment contains the application's local stack and heap.

Parameters *memory-options* is an optional keyword specifying whether the segment is fixed or moveable. It must be one of the following:

FIXED Segment remains at a fixed memory location

MOVEABLE Segment can be moved if necessary to compact memory

DISCARDABLE Segment can be discarded if no longer needed

The default is FIXED.

instance-options is an optional keyword that defines how many data segments are to be created for the module. It can be any one of the following:

NONE	No data segment
SINGLE	A single segment to be shared by all instances of the module (valid only for library modules)
MULTIPLE	One segment for each instance

The default is MULTIPLE.

Example

```
DATA MOVEABLE SINGLE
```

SEGMENT *segmentname memory-option pure-option load-option
minalloc*

Purpose This statement defines an additional code segment for applications that use a middle model memory organization.

Parameters *segmentname* is a character string naming the new segment. It can be any name, including the predefined segment names “_TEXT” or “_DATA” that represent the CODE and DATA segments. Redefining the CODE or DATA segment changes that segment’s default values.

memory-option is an optional keyword specifying whether the segment is fixed or moveable. It must be one of the following:

FIXED Segment remains at a fixed memory location

MOVEABLE Segment can be moved if necessary to compact memory

DISCARDABLE Segment can be discarded if no longer needed

The default is FIXED.

pure-option is an optional keyword specifying whether the segment is pure or impure. It must be one of the following:

PURE Segment contains code only

IMPURE Segment contains a mixture of code and data

The default is PURE.

load-option is an optional keyword specifying when the segment is to be loaded. It must be one of the following:

PRELOAD Segment is loaded immediately

LOADONCALL Segment is loaded when called

The default is PRELOAD.

minalloc is an optional integer number specifying the minimum number of bytes to be allocated for this segment.

Notes PURE and IMPURE are used for 286 protected mode programs.

Example

SEGMENTS

```
_TEXT FIXED  
_INIT PRELOAD MOVEABLE DISCARDABLE 15  
_RES PRELOAD MOVEABLE DISCARDABLE 8
```

EXPORTS *exportname ordinal-option res-option data-option parameter-option*

Purpose This statement defines the names and attributes of the functions to be exported to other applications. The **EXPORTS** keyword marks the beginning of the definitions. It can be followed by any number of export definitions, each on a separate line.

Parameters *exportname* is one or more ASCII characters defining the function name. It has the form:

entryname[=internalname]

where *entryname* is the name to be used by other applications to access the exported function, and *internalname* is an optional parameter that defines the actual name of the function if *entryname* is not the actual name.

ordinal-option is an optional parameter defining the function's ordinal value. It has the form:

@ordinal

where *ordinal* is an integer number specifying the function's ordinal value. The ordinal value defines the location of the function's name in the application's string table.

res-option is the optional keyword **RESIDENTNAME** specifying that the function's name must be resident at all times.

data-option is the optional keyword **NODATA** specifying that the function is not bound to a specific data segment. When invoked, the function uses the current data segment.

parameter-option is an optional integer number specifying the number of words the function expects to be passed as parameters.

Example

```
EXPORTS
    SampleRead=read2bin @1 8
    StringIn=str1 @2 4
    CharTest NODATA
```

IMPORTS *internal-option modulename entry-option*

Purpose This statement defines the names and attributes of the functions to be imported from other applications. The **IMPORTS** keyword marks the beginning of the definitions. It can be followed by any number of import definitions, each on a separate line.

Parameters *internal-option* is an optional parameter specifying the name that the application will use to call the function. It has the form:

internal-name=

where *internal-name* is one or more ASCII characters. This name must be unique.

modulename is the name of the executable module containing the function.

entry-option is an optional parameter specifying the function to be imported. It can be one of the following:

.*entryname*
.entryordinal

where *entryname* is the actual name of the function, and *entryordinal* is the ordinal value of the function.

Example

```
IMPORTS
    Sample.SampleRead
    write2hex=Sample.SampleWrite
    Read.1
```

STUB *filename*

Purpose This statement appends the old style executable file given by *filename* to the beginning of the module. The executable stub should display a warning message and terminate if the user attempts to execute the module without having loaded Windows. The default file WINSTUB.EXE can be used if no other actions are required.

Parameters *filename* is the name of the old style executable file to append to the module. The name must have the MS-DOS filename format.

6.4 Windows Initialization File

A Windows initialization file is a list of application names and option values that applications can access when executing. The Windows initialization file defines what initial values the user would like application-specific options to have. Each option has a corresponding keyname and value.

The Windows initialization file is kept in a special control file, named WIN.INI, that Windows searches for in the Windows boot directory (the current working directory at the time the user invoked Windows). If not found in the boot directory, Windows searches the directories given in the PATH environment variable, stopping with the first WIN.INI file it finds. If it finds the file, it reads its contents and performs the specified actions.

The WIN.INI file is an ordinary text file that you can create using a text editor. It has the general form

```
[application-name]
keyname = value
.
.
.
```

where [application-name] is the name of an application (usually the filename of the module file containing the application). The enclosing brackets ([]) are required. keyname = value is a special line defining the value of specific options associated with the application.

You can also place comments in a file. A comment may be placed on any line in which the first non-space character is a semicolon (;).

Applications can also access and change settings in WIN.INI during a Windows session. However, since WIN.INI is shared by all Windows applications, any application that makes a change to sections of WIN.INI that could affect other applications is responsible for notifying other applications of the change with the WM_WININICHANGE message.

You can give any number of application names. Each name can be followed by one or more keyname-and-value lines. A keyname-and-value line that appears immediately below an application name belongs to that application.

A keyname is the name of an option. It can consist of any combination of letters and digits, but must be followed immediately by the equal sign (=). A value is the option's value. It can be any one of the following:

Value	Description
Integer	Must use decimal digits.
String	Can be any combination of letters, digits, and punctuation. Leading spaces are ignored.
Quoted string	Can be any combination of letters, digits, and punctuation, but must be enclosed in double quotation marks. Spaces within the quotation marks are not ignored. A pair of quotation marks ("") in the string are treated as a single mark.

6.4.1 Windows Section

Syntax

```
[windows]
Run = list
Load = list
Device = output-device-name, device-driver, port-connection
DoubleClickSpeed=milliseconds
CursorBlinkRate=milliseconds
NullPort=null-portname
```

The **windows** section contains optional settings that take effect when Windows is first invoked. The **Run** and **Load** fields tell Windows to load a particular application or applications when Windows is first started. The *list* parameter is a list of one or more application filenames, separated by commas or whitespace. (Commas have a special meaning in the **Run** line, as described below.)

All applications listed in the **Load** line are run and placed in the icon area when Windows is first started. All applications listed in the **Run** line are run and displayed as windows when Windows is first started. A comma separating one application name from the next in the **Run** line means that the next application will be displayed in a new column. If only whitespace separates two application names in the **Run** line, the applications will be displayed in the same column.

The **Device** field defines the default output device. This is the device used for all printer output if no explicit device is given. The *output-device-name* can be any device name given in the **devices** section. An explicit port and driver must be assigned to the device. The user can set the printer mode in the control panel. *device-driver* is the module name (that is, the filename without the extension) of the device driver file. *port-name* is any port name given in the **ports** section.

The **DoubleClickSpeed** field sets the system's double click speed in milliseconds.

The **CursorBlinkRate** field sets the blinking rate (in milliseconds) for the system cursor.

The **NullPort** field sets the name used for a null port to *null-portname*. This name is used by the control panel and spooler (and other applications) in cases where a device is installed (that is, the device driver is present), but is not connected to any port.

6.4.2 Ports Section

Syntax

[ports]
portname:=baud-rate, parity, word-length, stop-bits

The **ports** section enumerates all available communication ports and defines default modes or settings. The *portname* must be the name of the communication port as it is recognized by MS-DOS.

The *baud-rate* gives the port's baud rate.

The *parity* setting is 'o', 'e', or 'n', for odd, even, or none, respectively.

The *word-length* field gives the length of a word, in bits.

The *stop-bits* field gives the number of stop bits to be used.

6.4.3 Colors Section

Syntax

[colors]
component=redvalue greenvalue bluevalue

The **Color** section defines the background color for the specified component of the Windows display screen. The *component* can be any one of the following:

Name	Component
Scrollbar	Scroll bar
Background	Icon area and screen background
ActiveTitle	Active caption bar
InactiveTitle	Inactive caption bar
Menu	Menu background
Window	Window client area background

WindowFrame	Title background and frame color
MenuText	Menu text
WindowText	Window text
TitleText	Title text

redvalue is an integer number specifying the intensity of red in the background. It can be any value from 0 to 255. *greenvalue* is an integer number specifying the intensity of green in the background. It can be any value from 0 to 255. *bluevalue* is an integer number specifying the intensity of blue in the background. It can be any value from 0 to 255.

Note

Windows expects a solid color for menu (MenuText), window (WindowText), and title text (TitleText) and for the window background (Window). The **GetNearestColor** function can be used to obtain the nearest system color.

6.4.4 Devices Section

Syntax

[devices]
device-name=driver-name, port-name [, portname...]

The **devices** section names the output devices that can be accessed by Windows device drivers and specifies the communications port or ports to which the devices are connected. If a device is not currently connected, the *portname* parameter should be the string specified in the **NullPort** field of the **windows** section of WIN.INI.

6.4.5 Fonts Section

Syntax

```
[fonts]
font-name==font-file
```

The **fonts** section describes one or more font files that are loaded by Windows when it is booted. *font-name* is the descriptive name of a font. *font-file* is the filename, without the extension, of a file containing font resources.

Note

An application can use the **EnumFonts** function to obtain information about the currently available fonts.

6.4.6 International Section

Syntax

```
[intl]
itemname==value
```

The **intl** section describes how to display dates, times, dollar amounts, and other items in countries other than the U.S. Windows does not require the **intl** section in U.S. versions of WIN.INI. Any application that uses the **intl** section must supply its own default value when calling **GetProfileString** to retrieve a value from this section.

The *itemname* and corresponding *value* can be any one of the following:

Name	Value
iCountry	Country Code (see the COUNTRY command in <i>MS-DOS 3.0 Reference Manual</i>)
iDate	0 for month-day-year 1 for day-month-year 2 for year-month-day

iCurrency	0 for currency symbol prefix, no separation 1 for currency symbol suffix, no separation 2 for currency symbol prefix, 1 character separation 3 for currency symbol suffix, 1 character separation
iDigits	Number of significant decimal digits in currency
iTime	0 for 12 hour clock 1 for 24 hour clock
iLzero	0 for no leading zeros 1 for leading zeros
s1159	Trailing string from 0:00 to 11:59
s2359	Trailing string from 12:00 to 23:59
sCurrency	Currency symbol string
sThousand	Thousands separator string
sDecimal	Decimal separator string
sDate	Date separator string
sTime	Time separator string
sList	List separator string

Chapter 7

Assembly Language Macros

7.1	Introduction	327
7.2	CMACROS.INC File	327
7.3	Cmacros Options	327
7.3.1	Memory Model Selection	328
7.3.2	Calling Conventions	329
7.3.3	Windows Prolog/Epilog	330
7.3.4	Stack Checking Option	330
7.4	Segment Macros	331
7.4.1	Storage Allocation Macros	334
7.4.2	Function Macros	337
7.4.3	Call Macros	341
7.4.4	Special Definition Macros	343
7.4.5	Error Macros	345
7.5	Using the Cmacros	346
7.5.1	Overriding Types	346
7.5.2	Symbol Redefinition	347
7.5.3	Cmacros: A Sample Function	347

(

)

)

7.1 Introduction

This chapter describes the **Cmacro** macros, a set of assembly-language macros that can be used with the Microsoft Macro Assembler (MASM) to create assembly-language Windows applications. The **Cmacros** provide a simplified interface to the function and segment conventions of high-level languages, such as C and Pascal.

The **Cmacros** are divided into the following groups:

- Segment Macros
- Storage Allocation Macros
- Function Macros
- Call Macros
- Special Definition Macros
- Error Macros

The following sections describe each group in detail.

7.2 CMACROS.INC File

The file CMACROS.INC contains the assembly-language definitions for all the **Cmacro** macros. This file must be included at the beginning of the assembly source file by using the **INCLUDE** directive. The line has the form:

```
INCLUDE cmacros.inc
```

A full pathname must be given if the macro file is not in the current working directory or not in a directory specified on the command line.

7.3 Cmacros Options

The **Cmacros** provide assembly-time options that define the memory model and the calling conventions to be used by the application. The options must be selected prior to the **INCLUDE** directive used to include the CMACROS.INC file.

7.3.1 Memory Model Selection

The memory model options specify the memory model to be used by the application. The memory model defines how many code and data segments are in the application. There are the following memory models:

Model	Definition
Small Model	One code and one data segment.
Middle Model	Multiple code segments and one data segment.
Large Model	Multiple code and data segments.
Compact Model	One code segment and multiple data segments.
Huge Model	Multiple code segments and multiple data segments with one or more data items larger than 64K bytes.

The memory option is selected by defining one of the following option names shown in Table 7.1 at the beginning of the assembly-language source file.

Table 7.1
Memory Options

Option Name	Memory Model	Code Size	Data Size
memS	small	small	small
memM	medium	large	small
memL	large	large	large
memC	compact	small	large
memH	huge	large	large

A name can be defined by using the **EQU** directive. The definition has the form:

memM EQU 1

If no option is selected, the default is "memS" or small model.

Selecting a memory model option results in the definition of two symbols that can be used for memory model dependent code:

SizeC	0 = small code	1 = large code
SizeD	0 = small data	1 = large data

7.3.2 Calling Conventions

The calling convention option specifies the high-level language calling convention to be used by the application. The calling convention is selected by defining the value of the symbol **?PLM**. The following table lists the values and conventions:

Table 7.2
Calling Conventions

?PLM value	Convention	Description
0	Standard C	The caller pushes the right-most argument onto the stack first, the left-most last. The caller pops the arguments off the stack after control is returned.
1	Pascal	The caller pushes the left-most argument onto the stack first, the right-most last. The called function pops the arguments off the stack.

The **?PLM** symbol value can be set using the **=** directive. The statement has the form:

?PLM = 1

The default is the Pascal convention.

7.3.3 Windows Prolog/Epilog

The Windows prolog/epilog option specifies whether or not special prologue and epilogue code should be used with each function. This special code defines the current data segment for the given function and is required for Windows applications.

This option is selected by defining the value of the symbol **?WIN**. The following table lists the values and conventions:

Table 7.3

Prolog/Epilog Code Options

?WIN value	Meaning
0	Disables the special prolog/epilog code.
1	Enables the special prolog/epilog code.

The **?WIN** symbol value can be set using the **=** directive. The statement has the form:

```
?WIN = 1
```

The default is the Windows prolog/epilog.

7.3.4 Stack Checking Option

Stack checking can be enabled by defining the symbol **?CHKSTK**. When enabled, the prologue code calls the externally defined routine **CHKSTK** to allocate local variables.

The **?CHKSTK** symbol can be defined using the **=** directive. The statement has the form:

```
?CHKSTK = 1
```

Once **CHKSTK** is defined, stack checking is enabled for the entire file.

The default (when **CHKSTK** is not defined) is no stack checking.

7.4 Segment Macros

The segment macros give access to the code and data segments to be used in an application. These segments have the names, attributes, classes, and groups required by Windows.

The **Cmacros** have two predefined segments, named **CODE** and **DATA**, that any application can use without special definition. Middle, large, and huge model applications can define additional segments using the **createSeg** macro.

createSeg *segName, logName, align, combine, class*

Purpose This macro creates a new segment having the specified name and segment attributes. The macro automatically creates an **assumes** macro and **OFFSET** equate for the new segment. This macro is intended to be used in middle model Windows applications to define non-resident segments.

Parameters *segName* is the actual name of the segment. This name is passed to the linker.

logName is the logical name of the segment. This name is used in all subsequent **sBegin**, **sEnd**, and **assumes** macros that refer to the segment.

align is the alignment type. It can be any one of the following:

BYTE
WORD
PARA
PAGE

combine is the combine type for the segment. It can be any one of the following:

PUBLIC
STACK
MEMORY
COMMON

If no combine type is given, a private segment is assumed.

class is the class name of the segment. The class name defines which segments must be loaded in consecutive memory.

Example

```
createSeg    _INIT, INITCODE, BYTE, PUBLIC, CODE  
sBegin     INITCODE  
assumes   CS:INITCODE  
  
            mov ax, initcodeOFFSET  sample  
  
sEnd      INITCODE
```

Notes The alignment, combine type, and class name are described in detail in the *Microsoft Macro Assembler Reference Manual*.

sBegin *segName*

Purpose This macro opens up a segment. It is similar to the **SEGMENT** assembler directive.

Arguments *segName* is the name of the segment to be opened. It can be one of the predefined segments, **CODE** or **DATA**, or the name of a user-defined segment.

Examples

```
sBegin DATA  
sBegin CODE
```

sEnd *segName*

Purpose This macro closes a segment. It is similar to the **ENDS** assembler directive.

Arguments *segName* is an optional name used for readability. If given, it must be the same name given in the matching **sBegin** macro.

Examples

```
sEnd  
sEnd DATA
```

assumes *segReg, segName*

Purpose This macro makes all references to data and code in the segment *segName* relative to the segment register given by *segReg*. It is similar to the **ASSUME** assembler directive.

Arguments *segReg* is the name of a segment register.

segName is the name of a predefined segment, CODE, or DATA, or a user-defined segment.

Examples

```
assumes CS, CODE
assumes DS, CODE
```

dataOFFSET *arg*

Purpose This macro generates an offset relative to the start of the group to which the **DATA** segment belongs. It is similar to the **OFFSET** assembler operator, but automatically provides the group name. For this reason, it should be used instead of **OFFSET**.

Parameters *arg* is a label name or offset value.

Example

```
mv ax,dataOFFSET label
```

codeOFFSET *arg*

Purpose This macro generates an offset relative to the start of the group to which the **CODE** segment belongs. It is similar to the **OFFSET** assembler operator, but automatically provides the group name. For this reason, it should be used instead of **OFFSET**.

Parameters *arg* is a label name or offset value.

Example

```
mv ax,codeOFFSET label
```

segNameOFFSET arg

Purpose This macro generates an offset relative to the start of the group to which the user-defined segment *segName* belongs. It is similar to the **OFFSET** assembler operator, but automatically provides the group name. For this reason, it should be used instead of **OFFSET**.

Parameters *arg* is a label name or offset value.

Example

```
mv ax,initcodeOFFSET label
```

7.4.1 Storage Allocation Macros

These macros allocate static memory (either private or public), declare externally defined memory and procedures, and allow the definition of public labels.

staticX name, initialValue, replication

Purpose This macro allocates private static memory storage.

Arguments *X* is the size of storage to be allocated. It can be any one of the following:

B	Byte
W	Word
D	Double word
Q	Quad word
T	Ten bytes
CP	Code Pointer (1 word for small and compact models)
DP	Data Pointer (1 word for small and middle models)

name is the reference name of the allocated memory.

initialValue is an optional initial value for the storage.

replication is an optional count of the number of times the allocation is to be duplicated. This argument generates the **DUP** assembler operator.

Examples

```
staticW flag,1  
staticB string, , 30
```

globalX *name, initialValue, replication*

Purpose This macro allocates public static memory storage.

Arguments **X** specifies the size of the storage to be allocated. It can be any one of the following:

B	Byte
W	Word
D	Double word
Q	Quad word
T	Ten bytes
CP	Code Pointer (1 word for small and compact models)
DP	Data Pointer (1 word for small and middle models)

name is the reference name of the allocated memory.

initialValue is the initial value for the storage.

replication is an optional count of the number of times the allocation is to be duplicated. This argument generates the **DUP** assembler operator.

Examples

```
globalW flag,1  
globalB string,0, 30
```

externX <namelist>

Purpose This macro defines one or more public (global) variables or functions.

Arguments X specifies the storage size or function type. It can be any one of the following:

B	Byte
W	Word
D	Double word
Q	Quad word
T	Ten bytes
CP	Code Pointer (1 word for small and compact models)
DP	Data Pointer (1 word for small and middle models)
NP	Near Function
FP	Far Function
P	Near for small and compact models; Far for other models

namelist is the name list of the variables or functions.

Examples

```
externB <DataBase>
externFP <SampleRead>
```

labelX <namelist>

Purpose This macro defines a label for one or more external variables or functions.

Arguments X specifies the storage size or function type. It can be any one of the following:

B	Byte
W	Word
D	Double word

Q	Quad word
T	Ten bytes
CP	Code Pointer (1 word for small and compact models)
DP	Data Pointer (1 word for small and middle models)
NP	Near Function
FP	Far Function
P	Near for small and compact models; Far for other models

namelist is the name list of the external variables or functions.

Examples

```
labelB <DataBase>
labelFP <SampleRead>
```

7.4.2 Function Macros

The function macros define the names, attributes, parameters, and local variables of functions.

cProc *procName*, <*attributes*>, <*autoSave*>

Purpose This macro defines the name and attributes of a function.

Arguments *procName* is the name of the function.

attributes specify the function type. They can be a combination of the following:

NEAR	A near function. It can only be called from the segment in which it is defined.
FAR	A far function. It can be called from any segment.
PUBLIC	A public function. It can be externally declared in other source files.

The default attribute is NEAR and private (i.e., cannot be declared externally in other source files). The NEAR and FAR attributes cannot be used together. If more than one attribute is selected, the angle brackets are required.

autoSave specifies a list of registers to be saved when the function is invoked and restored when exited. Any of the 8086's registers can be specified. If more than one register is listed, the angle brackets are required.

Examples

```
cProc proc1, <FAR>, <ds,es>
cProc proc2, <NEAR,PUBLIC>
cProc proc3,,ds
```

Notes The C calling conventions require that the SI and DI registers be saved if they will be altered.

The BP register is always saved, regardless of whether it is present in the *autoSave* list.

parmX <namelist>

Purpose This macro defines one or more function parameters. The parameters provide access to the arguments passed to the function. Parameters must appear in the same order as the arguments in the function call.

Arguments X specifies the storage size. It can be any one of the following:

- B Byte (allocated on a word boundary on the stack)
- W Word (allocated on a word boundary)
- D Double word (allocated on a word boundary)
- Q Quad word (aligned on a word boundary)
- T Ten byte word (aligned on a word boundary)
- CP Code Pointer (1 word for small and compact models)
- DP Data Pointer (1 word for small and middle models)

namelist is the name list of the parameters. If more than one parameter is given, the angle brackets are required.

Examples

```
ParmW var1
ParmB <var2,var3,var4>
ParmD <var5>
```

Notes

The **parmD** macro creates two additional symbols, **OFF_name** and **SEG_name**. **OFF_name** is the offset portion of the parameter; **SEG_name** is the segment portion.

Only the parameter name is required when referencing the corresponding argument.

Code: mov al, var1

Not: mov al, byte ptr var1 [bp]

LocalX <namelist>, size

Purpose

This macro defines one or more frame variables for the function. To maintain word alignment of the stack, the macro ensures that the total space allocated is an even number of bytes. It is assumed that the stack was on a word boundary upon entry into the function.

Arguments

X specifies the storage size. It can be any one of the following:

B	Byte (allocates a single byte of storage on the stack)
W	Word (allocated on a word boundary)
D	Double word (allocated on a word boundary)
V	Variable size (allocated on a word boundary)
Q	Quad word (aligned on a word boundary)
T	Ten byte word (aligned on a word boundary)
CP	Code pointer (same as used elsewhere)
DP	Data pointer (same as used elsewhere)

namelist is the name list of the frame variables for the function. If more than one parameter is given, the angle brackets are required.

size specifies the size of the variable. It is used with **localV** only.

Examples

```
LocalB <L1,L2,L3>
LocalW L4
LocalD <L5>
LocalV L6,%(size struc)
```

Notes

B type variables are not necessarily aligned on word boundaries.

The **localD** macro creates two additional symbols, **OFF_name** and **SEG_name**. **OFF_name** is the offset portion of the parameter; **SEG_name** is the segment portion.

Only the name is required when referencing a variable.

Code: mov al, var1

Not: mov al, byte ptr var1[bp]

cBegin *procName*

Purpose

This macro defines the actual entry point for the function **procName**. The macro creates code that sets up the frame and saves registers.

Arguments

procName is an optional function name. If given, it must be the same as given in the **cProc** macro immediately preceding the **cBegin** macro.

cEnd *procName*

Purpose

This macro defines the exit point for the function **procName**. The macro creates code that discards the frame, restores registers, and returns to the caller.

Arguments

procName is an optional function name. If given, it must be the same as given in the **cBegin** macro immediately preceding the **cEnd** macro.

Once a function has been defined using **cProc**, any formal parameters should be declared with the **parmX** macro and any local variables with the **localX** macro. The **cBegin** and **cEnd** macros must be used to delineate the code for the function. The following is an example of a complete function definition:

```

cProc    strcpy,<PUBLIC>,<si,di>
parmW    dst
parmW    src
localW   cnt

cBegin
    cld
    mov     si,src
    mov     di,dest
    push    ds
    pop    es
    xor    cx,cx
    mov    cnt,cx
loop:
    lodsb
    stosb
    inc    cnt
    cmp    al,0
    jnz    loop
    mov    ax,cnt
cEnd

```

7.4.3 Call Macros

The call macros can be used to call **cProc** functions and high-level language functions. These macros pass arguments according to the calling convention defined by the **?PLM** option.

cCall *procName*, <*argList*>, *underscores*

Purpose This macro pushes the arguments in the *argList* onto the stack, saves registers (if any), and calls the function *procName*.

Arguments *procName* is the name of the function to be called.

argList is an optional list of the names of arguments to be passed to the function. This list is not required if the **Arg** macro is used before the **cCall**. If more than one argument is given, the angle brackets are required.

underscores is an optional value that specifies whether or not an underscore should be prepended to *procName*. If this argument is blank, an underscore is prepended.

Examples

```
cCall    there,<pExt,ax,bx,pResult>
```

```
Arg      pExt
```

```
Arg      ax
```

```
cCall    there,<bx,pResult>
```

Notes

The arguments of an **Arg** macro are pushed onto the stack before any arguments in the *argList* of a **cCall** macro.

Byte type parameters are passed as words. There is no sign extension or zeroing of the high-order byte.

Immediate arguments are not supported.

Save <*regList*>

Purpose

This macro directs the next **cCall** macro to save the specified registers on the stack before calling a function, and to restore the registers after the function returns. The macro can be used to save registers that are destroyed by the called function.

The **Save** macro applies to one **cCall** macro only; each new **cCall** must have a corresponding **Save** macro. If two **Save** macros appear before a **cCall**, only the second macro is recognized.

Arguments

regList is a list of registers to be saved. If more than one register is given, the angle brackets are required.

Examples

```
Save    <cl,bh,si>
```

```
Save    <ax>
```

Arg <*nameList*>

Purpose

This macro defines the arguments to be passed to a function by the next **cCall** macro. The arguments are pushed onto the stack in the order given. This order must correspond with the order of the function parameters.

More than one **Arg** macro can be given before each **cCall**. Multiple **Arg** macros have the same effect as a single macro.

Arguments

nameList is a list of argument names to be passed to the function. All names must have been previously defined.

If more than one name is given, the angle brackets are required.

Examples

```
Arg      var1
Arg      var2
Arg      var3
Arg      <var1, var2, var3>
```

Notes

Byte type parameters are passed as words. There is no sign extension or zeroing of the high-order byte.

Immediate arguments are not supported.

7.4.4 Special Definition Macros

The special definition macros inform the **Cmacros** about user-defined variables, function register use, and register pointers.

DefX <namelist>*Purpose*

This macro registers the name of a user-defined variable with the **Cmacros**. Variables that are not defined using the **staticX**, **globalX**, **externX**, **parmX**, or **localX** macros cannot be referred to in other macros unless the name is registered, or the variable was defined with the **DW** assembler directive.

Arguments

X specifies the storage size of the variable. It can be any one of the following:

B	Byte
W	Word
D	Double word
Q	Quad word
T	Ten byte word
CP	Code pointer (same as used elsewhere)
DP	Data pointer (same as used elsewhere)

namelist is a list of variable names to be defined.

Example

```

maxSize db      132
        DefB    maxSize
dest      equ      wordptr es:[di]
        DefW    dest

```

RegPtr *name, segment, offset*

Purpose This macro defines a 32-bit value that is contained in the registers given by *segment* and *offset*. It allows the 32-bit value to be passed as a single argument in a **cCall** macro. If the **RegPtr** macro is not used, the result of the **cCall** macro is dependent on the calling conventions because the order in which the arguments are pushed could differ.

Arguments *name* is the name of the pointer to be created.
segment is the name of the register to contain the segment portion of the pointer.
offset is the name of the register to contain the offset portion of the pointer.

Example

```

RegPtr  destPtr,es,di
cCall   proc,<destPtr,ax>

```

FarPtr *name, segment, offset*

Purpose This macro defines a 32-bit pointer value that can be passed as a single argument in a **cCall** macro. Unlike the **RegPtr** macro, the *segment* and *offset* values do not have to be in registers.

Arguments *name* is the name of the pointer to be created.
segment is the text defining the segment portion of the pointer.
offset is the text defining the offset portion of the pointer.

Example

```

FarPtr  destPtr,es,<wordptr 3[si]>
cCall   proc,<destPtr,ax>

```

7.4.5 Error Macros

The error macros allow assertions to be coded into an assembly source program. This allows the coding of optimum instruction sequences for some operations based on variable allocation or bit position of a flag in a word, and asserting that the assumptions made are true.

Error macros generate an error message to the console and an error message in the listing. Both the text that caused the error, and the result of its evaluation, is displayed in the generated error message.

errnz <expression>

Purpose This macro evaluates a given expression. If the result is not zero, an error is displayed.

Arguments *expression* is the expression to be evaluated. The angle brackets are required if there are any spaces in the expression.

Examples

```
x      db      ?
Y      db      ?

mov     ax, word ptr x
errnz  <(OFFSET y) - (OFFSET x) -1>
```

If during assembly *x* and *y* receive anything but sequential storage locations, **errnz** displays an error message.

```
table1           struc
.
.
.
table1len       equ      $-table1
table1          ends

table2           struc
.
.
.
table2len       equ      $-table2
table2          ends

errnz  table1Len-table2Len
```

If during assembly the length of two tables is not the same, **errnz** displays an error message.

errn\$ label, bias

Purpose This macro subtracts the offset of *label* from the offset of the location counter, then adds *bias* to the result. If this result is not zero, then an error message is displayed.

Arguments *label* is a label corresponding to a memory location.
bias is a signed bias value. A plus or minus sign is required.

Example

```
;           end of previous code
errn$   function1
function1:
```

If a function originally located immediately after another piece of code is ever moved, **errn\$** displays an error message.

7.5 Using the Cmacros

This section explains the assembly-language statements generated by some of the **Cmacros** and illustrates their use with an example of a **Cmacros** function called “BITBLT.”

7.5.1 Overriding Types

Parameters and local variables created using the **parmX** and **localX** macros actually correspond to expressions of the form:

LocalB x	==>	x equ byte ptr [bp+nn]
ParmB y	==>	y equ byte ptr [bp+nn]

where *nn* is an offset from the current BP register value.

These expressions allow the usage of the names without having to explicitly type in “type ptr” and “[BP]+offset” operators. This means “x” can be referred to as

```
mov     al, x
```

and “y” as:

```
mov     ax, Y
```

A problem arises if the type must be overridden. The assembler creates an error message if it encounters the line:

```
mov ax,word ptr x
```

This can be solved by enclosing the name in parentheses:

```
mov ax,word ptr (x)
```

One exception to the above is the **LocalV** macro. The expression generated by this macro does not have a type associated with it. Therefore it can be overridden without the parentheses. For example:

```
LocalV horse,10 ==> horse equ [bp+nn]
```

7.5.2 Symbol Redefinition

Any symbol defined by a **parmX** in one function can be redefined as a parameter in any other function. This allows different functions to refer to the same parameter by the same name, regardless of its location on the stack.

7.5.3 Cmacros: A Sample Function

The following example defines an assembly function “BITBLT.” BITBLT is a FAR and PUBLIC type function. When BITBLT is invoked, the SI and DI registers are automatically saved, and automatically restored upon exit. Note that the BP register is always saved.

BITBLT is passed seven double word pointers on the stack. Space will be allocated on the stack for eight frame variables (one structure, five bytes, and two words).

The **cBegin** macro defines the start of the actual code. The *pExt* parameter is loaded, and some values are loaded into registers. The DS and SI registers are saved on the following **cCall**.

Another C function, “THERE”, is invoked by the **cCall** macro. Four arguments are passed to THERE: pDestBitmap , the 32-bit pointer in DS:SI, register AX, and register BX. The **cCall** macro places the arguments on the stack in the correct order.

When THERE returns, the arguments placed on the stack are automatically removed, and the DS and SI registers are restored.

When **cEnd** is reached, the frame variables are removed, any autosave registers are restored, and a return of the correct type (near or far) is performed.

Example

```
cProc BITBLT,<FAR,PUBLIC>,<si,di>

ParmD  pDestBitmap      ;--> to dest bitmap descriptor
ParmD  pDestOrg         ;--> to dest origin (a point)
ParmD  pSrcBitmap       ;--> to source bitmap descriptor
ParmD  pSrcOrg          ;--> to source origin
ParmD  pExt              ;--> to rectangle extent
ParmD  pRop              ;--> to rasterop descriptor
ParmD  pBrush             ;--> to a physical brush

LocalV  nOps,4           ;# of each operand used

LocalB  phaseH           ;Horizontal phase (rotate count)
LocalB  PatRow            ;Current row for patterns [0..7]
LocalB  direction         ;Increment/decrement flag

LocalW  startMask        ;mask for first dest byte
LocalW  lastMask          ;mask for last dest byte

LocalB  Firstfetch        ;Number of first fetches needed
LocalB  stepDirection     ;Direction of move (left right)

cBegin

lds    si,pExt
mov    ax,extentX[si]
mov    bx,extentY[si]

RegPtr dest,ds,si
Save   <ds,si>

cCall  THERE,<pDestBitmap,dest,ax,bx>

mov    extentX[si],cx
mov    extentY[si],dx

cEnd
```

Chapter 8

Window Messages

8.1	Introduction	351
8.2	Window Management Messages	352
8.3	Initialization Messages	362
8.4	Input Messages	364
8.5	System Messages	377
8.6	Clipboard Messages	381
8.7	System Information Messages	386
8.8	Control Messages	388
8.8.1	Button Control Messages	388
8.8.2	Edit Control Messages	390
8.8.3	List Box Messages	397
8.9	Notification Messages	401
8.9.1	Button Notification Codes	401
8.9.2	Edit Control Notification Codes	401
8.9.3	List Box Notification Codes	402
8.10	Scroll Bar Messages	402
8.11	Non-Client Area Messages	402

—

—

—

—

—

8.1 Introduction

Windows communicates with applications through formatted window messages. The messages are sent to an application's window function to let the function process the messages as desired.

A message consists of three parts: a message number, a word parameter, and a long parameter. Message numbers are identified by predefined message names. The names begin with "WM_" and suggest the meaning or origin of the message. The word and long parameters, named *wParam* and *lParam*, contain values that depend on the message number. If a given message does not use the parameter, it is set to 0.

The *lParam* parameter often contains more than one piece of information. For example, the high-order word may contain a handle to a window, and the low-order word an integer value. The HIWORD and LOWORD utility macros can be used to extract the high- and low-order words of *lParam*. The HIBYTE and LOBYTE utility macros can also be used with HIWORD and LOWORD to access any of the bytes. Casting can also be used.

There are four ranges of message numbers, as follows:

0 – WM_USER-1 Reserved for use by Windows.

WM_USER – 7FFF (hexadecimal)
“Integer” messages for use by applications.

8000 – BFFF (hexadecimal)
Reserved for use by Windows.

C000 – FFFF (hexadecimal)
“String” messages for use by applications.

Message numbers in the first range (0 – WM_USER-1) are defined by Windows and discussed in the remainder of this chapter. Values in this range that are not explicitly defined are reserved for future use by Windows.

Message numbers in the second range (WM_USER – 7FFF) can be defined and used by an application to send messages within the application. However, these messages should not be sent to other applications unless the applications have agreed in advance to exchange messages and to attach the same meaning to the message numbers.

Message numbers in the third range (8000 – BFFF) are reserved for future use by Windows.

Message numbers in the fourth range (C000 – FFFF) are defined when an application calls **RegisterWindowMessage** to obtain a message number for a string. All applications that register the identical string can use the associated message number for exchanging messages with each other. However, the actual message number is not a constant and cannot be assumed to be the same in different window sessions.

8.2 Window Management Messages

This section describes the messages that Windows sends to an application when the state of a window changes.

WM_CREATE

Purpose This message occurs when **CreateWindow** is called. It is sent before the **CreateWindow** call returns and before the window is made visible. The message informs the application that it can now perform any initialization desired.

Parameters *wParam* is not used.

lParam is a long pointer to a **CREATESTRUCT** data structure, whose fields correspond to the parameters passed to **CreateWindow**.

WM_SETVISIBLE

Purpose This message is sent immediately before a window is made visible or hidden.

Parameters *wParam* is nonzero if the window is being made visible, and zero if it is being made invisible.

lParam is not used.

WM_QUERYOPEN

Purpose This message is sent to an icon when the user is requesting that it be opened into a window. Returning zero prevents the icon from being opened; returning nonzero permits it.

Parameters *wParam* and *lParam* are not used.

Default Action **DefWindowProc** returns nonzero.

WM_ENABLE

- Purpose* This message occurs after a window has been enabled or disabled.
- Parameters* *wParam* is nonzero if the window has been enabled, and zero if it has been disabled.
lParam is not used.

WM_SETFOCUS

- Purpose* This message is sent after a window gets the input focus.
- Parameters* *wParam* contains the handle of the window losing the input focus (may be NULL).
lParam is not used.
- Notes* If an application wants to display a caret, it should call the appropriate caret functions at this point.

WM_KILLFOCUS

- Purpose* This message is sent immediately before a window loses the input focus.
- Parameters* *wParam* contains the handle of the window receiving the input focus (may be NULL).
lParam is not used.
- Notes* If an application is displaying a caret, the caret should be destroyed at this point.

WM_ACTIVATE

- Purpose* This message occurs when a window becomes active or inactive.
- Parameters* *wParam* specifies the new state of the window. It is zero if the window is inactive. It is one of the following nonzero values if the window is becoming active:

Value	Meaning
1	The window is becoming active through some other method than a mouse click (for example, a call to SetActiveWindow or selection of the window by the user through the keyboard interface.)
2	The window is becoming active because of a mouse click by the user. Any mouse button can be clicked: right, left, or middle.

The high-order word of *lParam* is nonzero if the window is iconic. Otherwise, it is zero. The value of the low-order word of *lParam* depends on the value of *wParam*. If *wParam* is zero, the low-order word of *lParam* is a handle to the window becoming active. If *wParam* is nonzero, the low-order word of *lParam* is the handle of the window becoming inactive (this handle may be NULL).

Default Action If the window is becoming active and it is not iconic, **DefWindowProc** sets the input focus to the window.

WM_ACTIVATEAPP

Purpose This message is sent when the window that is being activated belongs to a different application than the window that previously was active. If the activation is changing to another window created by the same application, no WM_ACTIVATEAPP message is sent.

Parameters *wParam* is nonzero if the window is being made active. If *wParam* is zero, the window is becoming inactive. If *wParam* is nonzero, the low-order word of *lParam* contains the task handle of the previously active application. If *wParam* is zero, the low-order word of *lParam* contains the task handle of the application being activated.

WM_SHOWWINDOW

Purpose This message is sent whenever a window is to be hidden or shown. A window is hidden or shown when one of the following events occurs:

1. **ShowWindow** is called.
2. A tiled window is zoomed or “unzoomed.”
3. A tiled or popup window is closed (made iconic) or opened (displayed on the screen).

When a tiled window is zoomed, all popup windows not associated with that tiled window are hidden (that is, they receive a WM_SHOWWINDOW message with *wParam* set to 0). When the window is unzoomed, all hidden popup windows are shown.

When a tiled window is closed, all popup windows associated with the window are hidden.

Parameters *wParam* is nonzero if the window is being shown, 0 if it is being hidden.

lParam is 0 if the message is sent because of a **ShowWindow** call. Otherwise, *lParam* is one of the following values:

Value	Meaning
SW_OTHERZOOM	Another window is being zoomed.
SW_OTHERUNZOOM	Another window is being unzoomed.
SW_PARENTCLOSING	The parent window is closing (being made iconic) or a popup window is being hidden.
SW_PARENTOPENING	The parent window is opening (being displayed) or a popup window is being shown.

Default Action **DefWindowProc** hides or shows the window as specified.

Notes

If an application has a hidden popup window when another window zooms and unzooms, the popup window will be shown when the window unzooms. To prevent this, the application should process any WM_SHOWWINDOW messages sent to the hidden popup window instead of passing them directly to **DefWindowProc**.

WM_SIZE

Purpose

This message occurs after the size of a window has been changed.

Parameters

wParam contains a value defining the type of resizing requested. It can be one of the following:

Value	Meaning
SIZEICONIC	Window has been made iconic.
SIZEFULLSCREEN	Window has been made full-screen.
SIZENORMAL	Window has been resized, but neither SIZEICONIC nor SIZEFULLSCREEN applies.
SIZEZOOMSHOW	Sent to all tiled windows when some other window has been "unzoomed" back to its former position.
SIZEZOOMHIDE	Sent to all tiled windows when some other window is zooming to full-screen.

lParam contains the new width and height of the client area of the window. The width is in the low-order word; the height is in the high-order word.

Notes

If **SetScrollPos** or **MoveWindow** is called for a child window as a result of the WM_SIZE message, the *bRedraw* parameter should be nonzero to force the window to be repainted.

WM_MOVE

Purpose This message is sent when a window is moved.

Parameters *wParam* is not used.

lParam contains the new location of the upper left corner of the client area of the window. The x coordinate is in the low-order word and the y coordinate is in the high-order word. For child style windows, the location is in the client coordinates of the parent window. For tiled and popup style windows, the location is in screen coordinates.

WM_ERASEBKGND

Purpose This message occurs when the window background needs erasing (for example, when a popup window is removed). It is sent in preparation for painting an invalidated region.

Parameters *wParam* contains the display context handle.

lParam is not used.

Default Action The background is erased, using the class background brush specified by the *hbrbackground* field in the class structure.

Return Value WM_ERASEBKGND returns nonzero if the background is erased. Otherwise, it returns zero. If the application processes the WM_ERASEBKGND message, it is responsible for returning the appropriate value.

Notes If *hbrbackground* is NULL, the application is responsible for processing the WM_ERASEBKGND message and erasing the background color. When processing the WM_ERASEBKGND message, the application must align the origin of the brush it intends to use with the window coordinates by first calling **UnrealizeObject** for the brush, then selecting the brush.

WM_PAINT

Purpose This message occurs whenever Windows or an application makes a request to repaint a portion of an application's window. The message is sent either when **UpdateWindow** is called or by **DispatchMessage** when the application obtains a WM_PAINT message using **GetMessage** or **PeekMessage**.

Parameters *wParam* is not used.

lParam contains a long pointer to a data structure containing information about the area of the screen to be painted. The data structure has **PAINTSTRUCT** type.

Default Action None.

WM_CTLCOLOR

Purpose This message is sent to the parent window of a predefined control or message box when the control or message box is about to be drawn. By responding to this message, the parent window can set the text and background colors of the child window, using the display context handle given in the *wParam* parameter.

Parameters *wParam* is a handle to the display context for the child window.

The low-order word of *lParam* is a handle to the child window. The high-order word is one of the following values, indicating the type of control:

Value	Control Type
CTL_MSGBOX	Message box
CTL_EDIT	Edit control
CTL_LISTBOX	List box control
CTL_BTN	Button control
CTL_DLG	Dialog box
CTL_SCROLLBAR	Scroll bar control
CTL_STATIC	Static control

Default Action **DefWindowProc** selects the default system colors.

Notes When processing the WM_CTLCOLOR message, the application must align the origin of the brush it intends to use with the window coordinates by first calling **UnrealizeObject** for the brush, then setting the brush origin to the upper left corner of the window.

WM_GETTEXT

- Purpose* This message is used to copy the text corresponding to a window. For edit controls, the text to be copied is the content of the control. For button controls, the text is the button name. For other windows, the text is the window caption.
- Parameters* *wParam* is an integer value specifying the maximum number of bytes to be copied, including the null terminator.
lParam contains a long pointer to the buffer to receive the text.
- Notes* This message returns the number of bytes copied.

WM_GETTEXTLENGTH

- Purpose* This message is used to find the length, in bytes, of the text associated with a window. The length does not include the null terminator. For edit controls, the text is the content of the control. For button controls, the text is the button name. For other windows, the text is the window caption.
- Parameters* *wParam* and *lParam* are not used.
- Notes* This message returns the length of the given text.

WM_SETTEXT

- Purpose* This message is used to set the text of a window. For edit controls, the text to be set is the content of the control. For button controls, the text is the button name. For other windows, the text is the window caption.
- Parameters* *wParam* is not used.
lParam contains a long pointer to a null-terminated string that is the window text.

WM_SETREDRAW

- Purpose* This message sets or clears the redraw flag, which determines whether updates to a control are displayed. When the redraw flag is set, the control is redrawn immediately after each change. When the redraw flag is cleared, no redrawing is done.

For example, this message could be sent to a list box to suppress update display before adding several items to the list, then sent again to restore update display after the items are added.

Parameters If *wParam* is nonzero, the redraw flag is set. If *wParam* is zero, the flag is cleared.
lParam is not used.

WM_GETDLGCODE

Purpose This message is sent by the Windows dialog manager to a control. Normally, the dialog manager handles all arrow key and tab key input to the control. By responding to the WM_GETDLGCODE message, an application can take control of a particular type of input and process the input itself.

Parameters *wParam* and *lParam* are not used.

Return Value If processing the WM_GETDLGCODE message, the application should return one of the following values, indicating which type of input it will handle itself:

Value	Type of input
DLG_WANTARROWS	Arrow keys
DLG_WANTTAB	Tab key
DLG_WANTALLKEYS	All keyboard input
DLG_HASSETSEL	EM_SETSEL messages

Default Action **DefWindowProc** returns 0.

Notes Although **DefWindowProc** always returns 0 in response to the WM_GETDLGCODE message, the window functions for the predefined control classes return a code appropriate for each class.

WM_CLOSE

Purpose This message occurs whenever the window is closed.

Parameters *wParam* is not used.

lParam is not used.

Default Action **DefWindowProc** calls the **DestroyWindow** function to destroy the window.

Notes An application can prompt the user for confirmation before destroying a window by processing the WM_CLOSE message and calling **DestroyWindow** only if the user confirms the choice.

WM_DESTROY

Purpose This message is sent when **DestroyWindow** is called, after the window has been removed from the screen. The WM_DESTROY message is sent to a parent window before any of its child windows are destroyed.

Parameters *wParam* and *lParam* are not used.

Notes If a window receiving the WM_DESTROY message is part of the clipboard viewer chain (set through a call to **SetClipboardViewer**), its first action must be to remove itself from the clipboard viewer chain by calling **ChangeClipboardChain**.

WM_QUERYENDSESSION

Purpose This message occurs when the user invokes the "End Session" command. If an application is willing to shut down, it should reply nonzero to this message. Otherwise, it returns zero.

If any application returns zero, the session is not ended. Windows stops sending WM_QUERYENDSESSION messages as soon as one application returns zero, and sends WM_ENDSESSION messages with *wParam* set to zero to any applications that have already returned nonzero.

Parameters *wParam* is not used.

lParam is not used.

Default Action **DefWindowProc** returns nonzero.

WM_ENDSESSION

<i>Purpose</i>	This message is sent to tell an application that has responded nonzero to a WM_QUERYENDSESSION message whether the session is actually being ended or not.
<i>Parameters</i>	<i>wParam</i> is nonzero if the session is being ended. Otherwise, it is zero. <i>lParam</i> is not used.
<i>Notes</i>	The application does not need to call DestroyWindow or PostQuitMessage when the session is being ended.

WM_QUIT

<i>Purpose</i>	This message indicates a request to terminate an application. It is generated by the application calling the PostQuitMessage function. It causes the GetMessage function to return zero.
<i>Parameters</i>	<i>wParam</i> contains the exit code given in the PostQuitMessage call. <i>lParam</i> is not used.

8.3 Initialization Messages

This section describes the window messages sent by Windows when a menu or dialog box is first created.

WM_INITMENUPOPUP

<i>Purpose</i>	This message is sent immediately before a popup menu is displayed. Processing this message allows an application to change the state of items on the popup menu before the menu is shown, without changing the state of the entire menu.
<i>Parameters</i>	<i>wParam</i> is the menu handle of the popup menu.

The low-order word of *lParam* is the index of the popup menu in the main menu. The high-order word of *lParam* is nonzero if the popup menu is the system menu. Otherwise, it is zero.

Default Action None.

WM_INITMENU

Purpose This message is a request to initialize a menu. It occurs when a user moves the mouse cursor into a menu bar and presses a mouse button, or presses a menu key. Windows sends the message before displaying the menu. This lets the application change the state of menu items, if desired, before the menu is shown.

Parameters *wParam* is the menu handle of the menu to be initialized.
lParam is not used.

Default Action None.

Notes A WM_INITMENU message is sent only when a menu is first accessed and only one WM_INITMENU is generated for each access. This means, for example, that dragging the mouse across several menu items while holding down the button does not generate new messages. The message does not provide information about menu items.

WM_INITDIALOG

Purpose This message is sent immediately before a dialog box is displayed. Processing this message allows an application to perform any initialization desired before the dialog box is made visible.

Parameters *wParam* contains a handle to the first control item in the dialog box that can be given the input focus. (Generally, this is the first item in the dialog box with WS_TABSTOP style). If the application returns a nonzero value in response to the WM_INITDIALOG message, Windows sets the input focus to the item given by *wParam*.

lParam is not used.

Notes When an application receives the WM_INITDIALOG message in a dialog box function, it can request that Windows set the input focus to the control given in *wParam* by returning nonzero. Returning zero tells Windows NOT to set

the focus; in this case, the dialog box function should set the input focus itself by calling **SetFocus** while processing the **WM_INITDIALOG** message.

If necessary, an application can call **EndDialog** during the processing of the **WM_INITDIALOG** message.

8.4 Input Messages

The messages in this section are sent when an application receives input through the mouse, keyboard, scroll bars, or system timer.

WM_MOUSEMOVE

Purpose This message occurs when the user moves the mouse.

Parameters *wParam* contains a value indicating whether or not various virtual keys are down. It can be any combination of the following:

Value	Meaning
MK_LBUTTON	Set if left button is down.
MK_RBUTTON	Set if right button is down.
MK_MBUTTON	Set if middle button is down.
MK_SHIFT	Set if SHIFT key is down.
MK_CONTROL	Set if CONTROL key is down.

lParam contains the x and y coordinates of the mouse cursor. The x coordinate is in the low-order word; the y-coordinate is in the high-order word. These coordinates are always relative to the top left corner of the window.

Default Action None.

Notes The MAKEPOINT macro can be used to convert the *lParam* parameter to a POINT.

WM_LBUTTONDOWN

- Purpose* This message occurs when the user presses the left mouse button.
- Parameters* *wParam* contains a value indicating whether or not various virtual keys are down. It can be any combination of the following:

Value	Meaning
MK_RBUTTON	Set if right button is down.
MK_MBUTTON	Set if middle button is down.
MK_SHIFT	Set if SHIFT key is down.
MK_CONTROL	Set if CONTROL key is down.

lParam contains the x and y coordinates of the mouse cursor. The x coordinate is in the low-order word; the y-coordinate is in the high-order word. These coordinates are always relative to the top left corner of the window.

Default Action None.

WM_LBUTTONUP

- Purpose* This message occurs when the user releases the left mouse button.
- Parameters* *wParam* contains a value indicating whether or not various virtual keys are down. It can be any combination of the following:

Value	Meaning
MK_RBUTTON	Set if right button is down.
MK_MBUTTON	Set if middle button is down.
MK_SHIFT	Set if SHIFT key is down.
MK_CONTROL	Set if CONTROL key is down.

lParam contains the x and y coordinates of the mouse cursor. The x coordinate is in the low-order word; the y-coordinate is in the high-order word. These coordinates are always relative to the top left corner of the window.

Default Action None.

WM_RBUTTONDOWN

Purpose This message occurs when the user presses the right mouse button.

Parameters *wParam* contains a value indicating whether or not various virtual keys are down. It can be any combination of the following:

Value	Meaning
MK_LBUTTON	Set if left button is down.
MK_MBUTTON	Set if middle button is down.
MK_SHIFT	Set if SHIFT key is down.
MK_CONTROL	Set if CONTROL key is down.

lParam contains the x and y coordinates of the mouse cursor. The x coordinate is in the low-order word; the y-coordinate is in the high-order word. These coordinates are always relative to the top left corner of the window.

Default Action None.

WM_RBUTTONUP

Purpose This message occurs when the user releases the right mouse button.

Parameters *wParam* contains a value indicating whether or not various virtual keys are down. It can be any combination of the following:

Value	Meaning
MK_LBUTTON	Set if left button is down.
MK_MBUTTON	Set if middle button is down.
MK_SHIFT	Set if SHIFT key is down.
MK_CONTROL	Set if CONTROL key is down.

lParam contains the x and y coordinates of the mouse cursor. The x coordinate is in the low-order word; the y-coordinate is in the high-order word. These coordinates are always relative to the top left corner of the window.

Default Action None.

WM_MBUTTONDOWN

Purpose This message occurs when the user presses the middle mouse button.

Parameters *wParam* contains a value indicating whether or not various virtual keys are down. It can be any combination of the following:

Value	Meaning
MK_LBUTTON	Set if left button is down.
MK_RBUTTON	Set if right button is down.
MK_SHIFT	Set if SHIFT key is down.
MK_CONTROL	Set if CONTROL key is down.

lParam contains the x and y coordinates of the mouse cursor. The x coordinate is in the low-order word; the y-coordinate is in the high-order word. These coordinates are always relative to the top left corner of the window.

Default Action None.

WM_MBUTTONDOWN

Purpose This message occurs when the user releases the middle mouse button.

Parameters *wParam* contains a value indicating whether or not various virtual keys are down. It can be any combination of the following:

Value	Meaning
MK_LBUTTON	Set if left button is down.
MK_RBUTTON	Set if right button is down.
MK_SHIFT	Set if SHIFT key is down.
MK_CONTROL	Set if CONTROL key is down.

lParam contains the x and y coordinates of the mouse cursor. The x coordinate is in the low-order word; the y-coordinate is in the high-order word. These coordinates are always relative to the top left corner of the window.

Default Action None.

WM_LBUTTONDOWNDBLCLK

Purpose This message occurs when the user double clicks the left mouse button.

Parameters *wParam* contains a value indicating whether or not various virtual keys are down. It can be any combination of the following:

Value	Meaning
MK_LBUTTON	Set if left button is down.
MK_RBUTTON	Set if right button is down.
MK_MBUTTON	Set if middle button is down.
MK_SHIFT	Set if SHIFT key is down.
MK_CONTROL	Set if CONTROL key is down.

lParam contains the x and y coordinates of the mouse cursor. The x coordinate is in the low-order word; the y-coordinate is in the high-order word. These coordinates are always relative to the top left corner of the window.

Default Action None.

Notes

Only windows whose window class has the CS_DBCLKS style can receive double click messages. Windows generates a double click message whenever the user presses, releases, and then presses a mouse button again within the system's double click time limit. Double clicking actually generates four messages: a down click message, an up click message, the double click message, and another up click message.

WM_RBUTTONDOWNDBLCLK

Purpose

This message occurs when the user double clicks the right mouse button.

Parameters

wParam contains a value indicating whether or not various virtual keys are down. It can be any combination of the following:

Value	Meaning
MK_LBUTTON	Set if left button is down.
MK_RBUTTON	Set if right button is down.
MK_SHIFT	Set if SHIFT key is down.
MK_CONTROL	Set if CONTROL key is down.

lParam contains the x and y coordinates of the mouse cursor. The x coordinate is in the low-order word; the y-coordinate is in the high-order word. These coordinates are always relative to the top left corner of the window.

Default Action None.

Notes

Only windows whose window class has the CS_DBCLKS style can receive double click messages. Windows generates a double click message whenever the user presses, releases, and then presses a mouse button again within the system's double click time limit. Double clicking actually generates three messages: a down click message, an up click message, and finally the double click message.

WM_MBUTTONDOWNDBLCLK

Purpose This message occurs when the user double clicks the middle mouse button.

Parameters *wParam* contains a value indicating whether or not various virtual keys are down. It can be any combination of the following:

Value	Meaning
MK_LBUTTON	Set if left button is down.
MK_RBUTTON	Set if right button is down.
MK_MBUTTON	Set if middle button is down.
MK_SHIFT	Set if SHIFT key is down.
MK_CONTROL	Set if CONTROL key is down.

lParam contains the x and y coordinates of the mouse cursor. The x coordinate is in the low-order word; the y-coordinate is in the high-order word. These coordinates are always relative to the top left corner of the window.

Default Action None.

Notes Only windows whose window class has the CS_DBCLKS style can receive double click messages. Windows generates a double click message whenever the user presses, releases, and then presses a mouse button again within the system's double click time limit. Double clicking actually generates four messages: a down click message, an up click message, the double click message, and another up click message.

WM_KEYDOWN

Purpose This message is sent when a non-system key is pressed.

Parameters *wParam* is an integer value specifying the virtual key code of the given key.

lParam contains the repeat count, scan code, key transition code, previous key state, and context code, as shown below. In the following list, bit 1 is the low-order bit.

Bit	Value
1-16	Repeat count (the number of times the keystroke is repeated as a result of the user holding down the key.)
17-25	Scan code (OEM-dependent value).
29	Context code (1 if ALT key was held down while key was pressed, 0 otherwise.)
30	Previous key state (1 if key was down before the message was sent, 0 if key was up).
31	Transition state (1 if key is being released, 0 if key is being pressed).

For WM_KEYDOWN messages, the key transition bit (bit 31) is 0 and the context code (bit 29) is 0.

Notes

A system key message occurs when any key is pressed while the ALT key is held down or while no window has the input focus; a non-system key message occurs when a key is pressed in any other circumstances.

Because of auto-repeat, more than one WM_KEYDOWN message may occur before a WM_KEYUP message is sent. The previous key state (bit 30) can be used to determine whether the WM_KEYDOWN message indicates the first down transition or a repeated down transition.

WM_KEYUP

Purpose

This message is sent when a non-system key is released.

Parameters

wParam is an integer value specifying the virtual key code of the given key.

lParam contains the repeat count, scan code, key transition code, previous key state, and context code, as shown below. In the following list, bit 1 is the low-order bit.

Bit	Value
1-16	Repeat count (the number of times the keystroke is repeated as a result of the user holding down the key.)
17-25	Scan code (OEM-dependent value).
29	Context code (1 if ALT key was held down while key was pressed, 0 otherwise.)
30	Previous key state (1 if key was down before the message was sent, 0 if key was up).
31	Transition state (1 if key is being released, 0 if key is being pressed).

For WM_KEYUP messages, the key transition bit (bit 31) is 1 and the context code (bit 29) is 0.

Notes A system key message occurs when any key is pressed while the ALT key is held down or while no window has the input focus; a non-system key message occurs when a key is pressed in any other circumstances.

WM_CHAR

Purpose This message is the result of a translated WM_KEYUP or WM_KEYDOWN message. It contains the ASCII value of the keyboard key being pressed or released.

Parameters *wParam* contains the ASCII value of the key.

lParam contains the repeat count, scan code, key transition code, previous key state, and context code, as shown below. In the following list, bit 1 is the low-order bit.

Bit	Value
1-16	Repeat count (the number of times the keystroke is repeated as a result of the user holding down the key.)
17-25	Scan code (OEM-dependent value).

- ()
- | | |
|----|--|
| 29 | Context code (1 if ALT key was held down while key was pressed, 0 otherwise.) |
| 30 | Previous key state (1 if key was down before the message was sent, 0 if key was up). |
| 31 | Transition state (1 if key is being released, 0 if key is being pressed). |

Default Action None.

Notes Since there is not necessarily a one-to-one correspondence between keys pressed and character messages generated, the information in the high-order word of *lParam* is not generally useful to applications. The information applies only to the most recent WM_KEYUP or WM_KEYDOWN message before the character message was posted.

WM_DEADCHAR

Purpose This message is the result of a translated WM_KEYUP or WM_KEYDOWN message. It specifies the character value of a dead key. A dead key is a key such as the umlaut (double dot) character that is combined with other characters to form a composite character. For example, the umlaut-O character consists of the dead key, umlaut, and the "O" key.

Parameters *wParam* contains the dead key character value.

lParam contains the repeat count, scan code, key transition code, previous key state, and context code, as shown below. In the following list, bit 1 is the low-order bit.

Bit	Value
1-16	Repeat count (the number of times the keystroke is repeated as a result of the user holding down the key.)
17-25	Scan code (OEM-dependent value).

- | | |
|----|--|
| 29 | 1 if ALT key was held down while key was pressed, 0 otherwise. |
| 30 | Previous key state (1 if key was down before the message was sent, 0 if key was up). |
| 31 | Transition state (1 if key is being released, 0 if key is being pressed). |

Default Action None.

- Notes* The WM_DEADCHAR message is typically used by applications that want to give the user feedback about each key pressed. Since there is not necessarily a one-to-one correspondence between keys pressed and character messages generated, the information in the high-order word of *lParam* is not generally useful to applications. The information applies only to the most recent WM_KEYUP or WM_KEYDOWN message before the character message was posted.

WM_TIMER

- Purpose* This message occurs when the time limit set for a given timer has elapsed.
- Parameters* *wParam* contains the timer ID, an integer value that uniquely identifies the timer.
lParam contains the long pointer to a function that was passed to **SetTimer** when the timer was created. If *lParam* is not NULL, the specified function is called directly, instead of sending the WM_TIMER message to the window function.

Default Action None.

WM_COMMAND

- Purpose* This message occurs when the user selects an item from a menu, when a control passes a message to its parent window, or when an accelerator keystroke is translated.
- Parameters* *wParam* contains either the menu item, the control ID, or the accelerator ID.
lParam contains 0 if the message is from a menu. If the message is an accelerator message, the high-order word of *lParam* is 1. If the message is from a control, the high-order

word of *lParam* contains the window handle of the control sending the message and the low-order word contains the control ID.

Notes

Accelerator keystrokes that are defined to select items from the system menu are translated into WM_SYSCOMMAND messages.

If an accelerator keystroke corresponding to a menu item occurs when the window owning the menu is iconic, no WM_COMMAND message is sent. However, if an accelerator keystroke occurs that does not match any of the items on the window's menu or on the system menu, a WM_COMMAND message is sent, even if the window is iconic.

WM_VSCROLL

Purpose

This message occurs whenever the user clicks the mouse on the vertical scroll bar.

Parameters

wParam contains a scroll bar code specifying the user's scrolling request. It can be any one of the following:

Value	Meaning
SB_LINEUP	Scroll one line up.
SB_LINEDOWN	Scroll one line down.
SB_PAGEUP	Scroll one page up.
SB_PAGEDOWN	Scroll one page down.
SB_THUMBPOSITION	Scroll to absolute position.
SB_THUMBTRACK	Thumb is being dragged and is current at the specified position.
SB_TOP	Scroll to top.
SB_BOTTOM	Scroll to bottom.
SB_ENDSCROLL	End of scroll.

lParam contains the current position of the thumb if *wParam* is either SB_THUMBPOSITION or SB_THUMBTRACK. Otherwise, *lParam* is not used. The thumb position, when given, is in the low-order word of *lParam*.

Notes The SB_THUMBTRACK message is typically used by applications that give some feedback while the thumb is being dragged.

If an application scrolls the document in the window, it must also reset the position of the thumb by using the **SetScrollPos** function.

WM_HSCROLL

Purpose This message occurs whenever the user clicks the mouse on the horizontal scroll bar.

Parameters *wParam* contains a scroll bar code specifying the user's scrolling request. It can be any one of the following:

Value	Meaning
SB_LINEUP	Scroll one line left.
SB_LINEDOWN	Scroll one line right.
SB_PAGEUP	Scroll one page left.
SB_PAGEDOWN	Scroll one page right.
SB_THUMBPOSITION	Scroll to absolute position.
SB_THUMBTRACK	Thumb is being dragged and is currently at the specified position.
SB_TOP	Scroll to far left.
SB_BOTTOM	Scroll to far right.
SB_ENDSCROLL	End of scroll.

lParam contains the current position of the thumb if *wParam* is either SB_THUMBPOSITION or SB_THUMBTRACK. Otherwise, *lParam* is not used. The thumb position, when given, is in the low-order word of *lParam*.

Notes The SB_THUMBTRACK message is typically used by applications that give some feedback while the thumb is being dragged.

If an application scrolls the document in the window, it must also reset the position of the thumb by using the **SetScrollPos** function.

8.5 System Messages

This section describes messages that Windows sends to an application when a user accesses a window's system menu, scroll bars, or size box. Although an application can process these messages, if desired, most applications pass them to the **DefWindowProc** function for default processing.

WM_SYSKEYDOWN

Purpose This message occurs whenever the user holds down the ALT key and then presses another key. It also occurs when no window currently has the input focus; in this case, the WM_SYSKEYDOWN message is sent to the active window. The window receiving the message can distinguish between these two contexts by checking the context code in *lParam*.

Parameters *wParam* contains the virtual key code of the key being pressed.

lParam contains the repeat count, scan code, key transition code, previous key state, and context code, as shown below. In the following list, bit 1 is the low-order bit.

Bit	Value
1-16	Repeat count (the number of times the keystroke is repeated as a result of the user holding down the key.)
17-25	Scan code (OEM-dependent value).
29	Context code (1 if ALT key was held down while key was pressed, 0 otherwise.)
30	Previous key state (1 if key was down before the message was sent, 0 if key was up).
31	Transition state (1 if key is being released, 0 if key is being pressed).

For WM_SYSKEYDOWN messages, the key transition bit (bit 31) is 0. The context code (bit 29) is 1 if the ALT key was down while the key was pressed, and is 0 if the message was sent to the active window because no window has the input focus.

Notes

When the context code is zero, the message can be passed to **TranslateAccelerator**, which will handle it as though it were a normal key message instead of a system key message. This allows accelerator keys to be used with the active window even if the active window doesn't have the focus.

Because of auto-repeat, more than one WM_SYSKEYDOWN message may occur before a WM_SYSKEYUP message is sent. The previous key state (bit 30) can be used to determine whether the WM_SYSKEYDOWN message indicates the first down transition or a repeated down transition.

WM_SYSKEYUP*Purpose*

This message occurs whenever the user releases a key that was pressed while the ALT key was held down. It also occurs when no window currently has the input focus; in this case, the WM_SYSKEYUP message is sent to the active window. The window receiving the message can distinguish between these two contexts by checking the context code in *lParam*.

Parameters

wParam contains the virtual key code of the key being released.

lParam contains the repeat count, scan code, key transition code, previous key state, and context code, as shown below. In the following list, bit 1 is the low-order bit.

Bit	Value
1-16	Repeat count (the number of times the keystroke is repeated as a result of the user holding down the key.)
17-25	Scan code (OEM-dependent value).
29	Context code (1 if ALT key was held down while key was pressed, 0 otherwise.)
30	Previous key state (1 if key was down before the message was sent, 0 if key was up).
31	Transition state (1 if key is being released, 0 if key is being pressed).

For WM_SYSKEYUP messages, the key transition bit (bit 31) is 1. The context code (bit 29) is 1 if the ALT key was

down while the key was pressed, and is 0 if the message was sent to the active window because no window has the input focus.

- Notes* When the context code is zero, the message can be passed to **TranslateAccelerator**, which will handle it as though it were a normal key message instead of a system key message. This allows accelerator keys to be used with the active window even if the active window doesn't have the focus.

WM_SYSCHAR

- Purpose* This message is the result of translating a WM_SYSKEYUP or WM_SYSKEYDOWN message. It specifies the virtual key code of the system menu key.
- Parameters* *wParam* contains the virtual key code of a system menu key. *lParam* contains the repeat count, scan code, key transition code, previous key state, and context code, as shown below. In the following list, bit 1 is the low-order bit.

Bit	Value
1-16	Repeat count (the number of times the keystroke is repeated as a result of the user holding down the key.)
17-25	Scan code (OEM-dependent value).
29	Context code (1 if ALT key was held down while key was pressed, 0 otherwise.)
30	Previous key state (1 if key was down before the message was sent, 0 if key was up).
31	Transition state (1 if key is being released, 0 if key is being pressed).

- Notes* When the context code is zero, the message can be passed to **TranslateAccelerator**, which will handle it as though it were a normal key message instead of a system key message. This allows accelerator keys to be used with the active window even if the active window doesn't have the focus.

Default Action None.

WM_SYSDEADCHAR

Purpose This message is the result of a translated WM_SYSKEYUP or WM_SYSKEYDOWN message. It specifies the character value of a dead key.

Parameters *wParam* contains the dead key character value.

lParam contains a repeat count and an auto-repeat count. The low-order word contains the repeat count; the high-order word contains the auto-repeat count.

Default Action None.

WM_SYSCOMMAND

Purpose This message occurs when the user selects a command from the system menu.

Parameters *wParam* is an integer value specifying the type of system command requested. It can be any one of the following:

Value	Meaning
SC_SIZE	Request to size the window
SC_MOVE	Request to move the window
SC_ICON	Request to make the window iconic
SC_ZOOM	Request to zoom the window
SC_CLOSE	Request to close the window
SC_NEXTWINDOW	Request to move to the next window
SC_PREVWINDOW	Request to move to the previous window
SC_VSCROLL	Request for a vertical scroll
SC_HSCROLL	Request for a horizontal scroll
SC_MOUSEMENU	Request for a menu through a mouse click
SC_KEYMENU	Request for a menu through a keystroke

lParam is not used.

Default Action **DefWindowProc** carries out the system menu request for the predefined actions specified above.

Notes In WM_SYSCOMMAND messages, the four low-order bits of *wParam* are used internally by Windows. When an application tests the value of *wParam*, it must combine the value FFF0 (hexadecimal) with the *wParam* value, using the bit-wise AND operator, to obtain the correct result.

The menu items in a system menu can be modified by using the **GetSystemMenu** and **ChangeMenu** functions. Applications that modify the system menu must process WM_SYSCOMMAND messages. Any WM_SYSCOMMAND messages not handled by the application must be passed to **DefWindowProc**. Any command values added by an application must be processed by the application and cannot be passed to **DefWindowProc**.

An application can carry out any system command at any time by passing a WM_SYSCOMMAND message to **DefWindowProc**.

Accelerator keystrokes that are defined to select items from the system menu are translated into WM_SYSCOMMAND messages; all other accelerator keystrokes are translated into WM_COMMAND messages.

8.6 Clipboard Messages

This section describes messages that an application may receive from applications trying to access the Windows clipboard.

WM_RENDERFORMAT

Purpose This message is a request to the clipboard owner to format the data last copied to the clipboard in the specified format and to pass a handle to the formatted data to the clipboard.

Parameters *wParam* specifies the data format. It can be any one of the formats described with the **SetClipboardData** function.
 lParam is not used.

Default Action None.

WM_RENDERALLFORMATS

Purpose This message is sent to the application that owns the clipboard when the application is being destroyed. The application should render the clipboard data in all formats it is capable of generating and pass a handle to each format to **SetClipboardData**. This ensures that the data in the clipboard can be rendered even though the application has been destroyed.

WM_DESTROYCLIPBOARD

Purpose This message is sent to the clipboard owner when the clipboard is emptied through a call to **EmptyClipboard**.

Parameters *wParam* and *lParam* are not used.

WM_DRAWCLIPBOARD

Purpose This message is sent to the first window in the clipboard viewer chain whenever the contents of the clipboard change. Only applications that have joined the clipboard viewer chain by calling **SetClipboardViewer** need to process this message.

Parameters *wParam* and *lParam* are not used.

Notes Each window receiving the WM_DRAWCLIPBOARD message is responsible for calling **SendMessage** to pass on the message to the next window in the clipboard viewer chain. The handle of the next window is returned by **SetClipboardViewer**; it may be modified in response to a WM_CHANGECBCHAIN message.

WM_CHANGECBCHAIN

Purpose This message notifies the first window in the clipboard viewer chain that a window is being removed from the clipboard viewer chain.

<i>Parameters</i>	<i>wParam</i> contains the handle to the window being removed from the clipboard viewer chain.
	The low-order word of <i>lParam</i> contains the handle to the window following the window being removed in the clipboard viewer chain.
<i>Notes</i>	Each window receiving the WM_CHANGECHAIN message is responsible for calling SendMessage to pass on the message to the next window in the clipboard viewer chain. If the window being removed is the next window in the chain, the window specified by the low-order word of <i>lParam</i> becomes the next window, and clipboard messages should be passed on to it.

WM_PAINTCLIPBOARD

<i>Purpose</i>	This message is sent when the clipboard contains a data handle for the CF_OWNERDISPLAY format (i.e., the clipboard owner is responsible for displaying the clipboard contents) and the clipboard application's client area needs repainting. The WM_PAINTCLIPBOARD message is sent to the owner of the clipboard to request repainting of all or part of the clipboard application's client area.
<i>Parameters</i>	<i>wParam</i> is a handle to the clipboard application window. <i>lParam</i> is a long pointer to a PAINTSTRUCT data structure defining what part of the client area to paint.
<i>Notes</i>	To determine whether the entire client area needs repainting or just a portion of it, the clipboard owner must compare the dimensions of the drawing area given in the rcPaint field of the PAINTSTRUCT structure to the dimensions given in the most recent WM_SIZECLIPBOARD message.

WM_SIZECLIPBOARD

<i>Purpose</i>	This message is sent when the clipboard contains a data handle for the CF_OWNERDISPLAY format (i.e., the clipboard owner is responsible for displaying the clipboard contents) and the clipboard application window has changed size.
<i>Parameters</i>	<i>wParam</i> is a handle to the clipboard application window.

The low-order word of *lParam* is a pointer to a **RECT** data structure specifying the area in which the clipboard owner should paint. The high-order word is not used.

Notes A WM_SIZECLIPBOARD message is sent with a null rectangle (0,0,0,0) as the new size when the clipboard application is about to be destroyed or made iconic. This permits the clipboard owner to free its display resources.

WM_VSCROLLCLIPBOARD

Purpose This message is sent when the clipboard contains a data handle for the CF_OWNERDISPLAY format (i.e., the clipboard owner is responsible for displaying the clipboard contents) and an event occurs in the clipboard application's vertical scroll bar.

Parameters *wParam* contains a handle to the clipboard application window.

The low-order word of *lParam* contains one of the following scroll bar codes:

Value	Meaning
SB_LINEUP	Scroll one line up
SB_LINEDOWN	Scroll one line down
SB_PAGEUP	Scroll one page up
SB_PAGEDOWN	Scroll one page down
SB_THUMBUPOSITION	Scroll to absolute position
SB_TOP	Scroll to top
SB_BOTTOM	Scroll to bottom
SB_ENDSCROLL	End of scroll

The high-order word of *lParam* contains the thumb position if the scroll bar code is SB_THUMBUPOSITION. Otherwise, the high-order word is not used.

Notes The clipboard owner should use **InvalidateRect** or repaint as desired. The scroll bar position should also be reset.

WM_HSCROLLCLIPBOARD

Purpose This message is sent when the clipboard contains a data handle for the CF_OWNERDISPLAY format (i.e., the clipboard owner is responsible for displaying the clipboard contents) and an event occurs in the clipboard application's horizontal scroll bar.

Parameters *wParam* contains a handle to the clipboard application window.

The low-order word of *lParam* contains one of the following scroll bar codes:

Value	Meaning
SB_LINEUP	Scroll one line left
SB_LINEDOWN	Scroll one line right
SB_PAGEUP	Scroll one page left
SB_PAGEDOWN	Scroll one page right
SB_THUMBPOSITION	Scroll to absolute position
SB_TOP	Scroll to far left
SB_BOTTOM	Scroll to far right
SB_ENDSCROLL	End of scroll

The high-order word of *lParam* contains the thumb position if the scroll bar code is SB_THUMBPOSITION. Otherwise, the high-order word is not used.

Notes The clipboard owner should use **InvalidateRect** or repaint as desired. The scroll bar position should also be reset.

WM_ASKCBFORMATNAME

Purpose This message is sent when the clipboard contains a data handle for the CF_OWNERDISPLAY format (i.e., the clipboard owner is responsible for displaying the clipboard contents) to request a copy of the format name.

Parameters *wParam* contains an integer value specifying the maximum number of bytes to copy.

lParam is a long pointer to the buffer where the copy of the format name is to be stored.

<i>Notes</i>	The clipboard owner should copy the name of the CF_OWNERDISPLAY type into the specified buffer, not exceeding the maximum number of bytes.
--------------	--

8.7 System Information Messages

The messages described in this section are sent when an application or the user makes a system-wide change that may affect other applications.

WM_SYSCOLORCHANGE

Purpose This message is sent to all top-level windows when a change is made in the system color setting.

Parameters *wParam* and *lParam* are not used.

Default Action Portions of the screen that are affected by the system color changes will receive WM_PAINT messages.

Notes If an application has brushes corresponding to the system colors, it should query the system for the new system colors and then recreate the brushes when this message is received.

WM_DEVMODECHANGE

Purpose This message is sent to all top-level windows whenever the user changes device mode settings.

Parameters *wParam* is not used

lParam is a long pointer to the device name as specified in the WIN.INI file.

Default Action None.

WM_FONTCHANGE

Purpose This message occurs when the pool of font resources changes. Any application that adds or removes fonts from the system (for example, through **AddFontResource** or **RemoveFontResource**) is responsible for sending this message to all top-level windows.

Parameters *wParam* and *lParam* are not used.

Default Action None.

Notes To send the WM_FONTCHANGE message to all top-level windows, an application can call **SendMessage** with the *hWnd* parameter set to FFFF (hexadecimal).

WM_TIMECHANGE

Purpose This message occurs when an application makes a change (or set of changes) to the system time. Any application that changes the system time is responsible for sending this message to all top-level windows.

Parameters *wParam* and *lParam* are not used.

Default Action None.

Notes To send the WM_TIMECHANGE message to all top-level windows, an application can use **SendMessage** with the *hWnd* parameter set to FFFF (hexadecimal).

WM_WININICHANGE

Purpose This message occurs when the Windows initialization file (WIN.INI) changes. Any application that makes a change to WIN.INI is responsible for sending this message to all top-level windows.

Parameters *wParam* is not used.

lParam contains a long pointer to a string specifying the name of the section that has changed (not including the square brackets). If more than one section has changed, *lParam* is zero, and the application receiving the message must find and handle the changes itself.

Default Action None.

Notes To send the WM_WININICHANGE message to all top-level windows, an application can use **SendMessage** with the *hWnd* parameter set to FFFF (hexadecimal).

WM_SYSTEMERROR

Purpose This message is sent to all top-level windows when an out-of-memory system error occurs.

Parameters *wParam* is the value 8, indicating an MS-DOS out-of-memory error.

lParam is not used.

Default Action The MS-DOS Executive window alerts the user to the out-of-memory condition.

Notes Applications do not need to process this message, since the out-of-memory condition is handled by the MS-DOS Executive.

8.8 Control Messages

Control messages are predefined window messages that direct a control to carry out a specified task. Applications send control messages to a control using the **SendMessage** function. The control carries out the specified task and returns a value indicating the result. The following sections describe the control messages for each control class.

Note

There are no predefined scroll bar control messages. Instead, the scroll bar controls use the same notification messages as scroll bars that are not controls, except that they notify their parent.

There are no static control messages.

8.8.1 Button Control Messages

This section describes messages that an application can send to a button control.

BM_GETCHECK

Purpose This message determines whether a radio button or check box is checked.

Parameters *wParam* and *lParam* are not used.

Return Value BM_GETCHECK returns nonzero if the radio button or check box is checked. Otherwise, BM_GETCHECK returns zero. It always returns zero for a push button.

BM_GETSTATE

Purpose This message returns nonzero if the cursor is over the button and the user is pressing a mouse button or the space bar.

Parameters *wParam* and *lParam* are not used.

Return Value BM_GETCHECK returns nonzero if the cursor is over the button and the user is pressing a mouse button or the space bar. Otherwise, BM_GETCHECK returns zero. For push buttons, it returns nonzero if the button is highlighted and zero if not.

BM_SETCHECK

Purpose This message checks or unchecks a radio button or check box.

Parameters If *wParam* is nonzero, a check is placed beside the button or box. If zero, the check (if any) is removed. For 3-state buttons, if *wParam* is 1, a check is placed beside the button. If *wParam* is 1, the button is grayed. If *wParam* is 0, the button is returned to its normal state (no checkmark or graying).

lParam is not used.

Return Value None.

Notes BM_SETCHECK has no effect on push buttons.

BM_SETSTATE

Purpose This message highlights a button or check box.

Parameters If *wParam* is nonzero, a black frame is drawn around the button, or the check box is emboldened. If zero, the highlight is removed.

Return Value None.

Notes Push buttons cannot be highlighted.

8.8.2 Edit Control Messages

This section describes the messages that an application can send to an edit control. In addition to the messages described below, the WM_SETFOCUS, WM_KILLFOCUS, WM_SETTEXT, WM_GETTEXT, WM_GETTEXTLENGTH, WM_SETREDRAW, and WM_ENABLE window messages can also be used.

EM_GETSEL

Purpose This message returns the starting and ending character positions of the current selection.

Parameters *wParam* and *lParam* are not used.

Return Value The long value returned contains the starting position in the low-order word. The high-order word contains the position of the first non-selected character after the end of the selection.

EM_SETSEL

Purpose This message selects all characters in the current text that are within the starting and ending character positions given by the *lParam* parameter.

Parameters *wParam* is not used.

lParam contains the starting position in the low-order word, and the ending position in the high-order word. The position values 0 to 32767 select the entire string.

Return Value None.

EM_GETRECT

Purpose This message retrieves the formatting rectangle of the control.

Parameters *wParam* is not used.

lParam must be a long pointer to a data structure with RECT type. The control copies the dimensions to the structure.

Return Value None.

EM_SETRECT

- Purpose* This message sets the formatting rectangle for a control. The text is reformatted and redisplayed to reflect the changed rectangle.
- Parameters* *wParam* is not used.
lParam must be a data structure with **RECT** type specifying the new dimensions of the rectangle.
- Return Value* None.
- Notes* This message will not be processed by single line edit controls.

EM_SETRECTNP

- Purpose* This message is the same as EM_SETRECT, except that the control is NOT repainted. Any subsequent repaints cause the control to be repainted to reflect the changed formatting rectangle. This message is used when the field is to be repainted later.
- Parameters* *wParam* is not used.
lParam must be a data structure having **RECT** type specifying the new dimensions of the rectangle.
- Return Value* None.
- Notes* This message will not be processed by single line edit controls.

EM_SETHANDLE

- Purpose* This message establishes the text buffer used to hold the contents of the control window.
- Parameters* *wParam* contains a handle to the buffer. The handle must be a local handle to a location in the application's data segment (DS). The edit control uses this buffer to store the currently displayed text instead of allocating its own buffer. If necessary, the control reallocates this buffer.
lParam is not used.

Return Value None.

Notes This message will not be processed by single line edit controls.

EM_GETHANDLE

Purpose This message returns the data handle of the buffer used to hold the contents of the control window. The handle is always a local handle to a location in the application's data segment.

Parameters *wParam* and *lParam* are not used.

Return Value None.

Notes This message will not be processed by single line edit controls.

EM_GETLINECOUNT

Purpose This message returns the number of lines of text in the edit control.

Parameters *wParam* and *lParam* are not used.

Return Value EM_GETLINECOUNT returns the number of lines of text in the control.

Notes This message will not be processed by single line edit controls.

EM_LINEINDEX

Purpose This message returns the number of character positions that occur before the first character in a given line.

Parameters *wParam* contains the desired line number, where the first line number is 0. If *wParam* is -1, the current line (the line containing the caret) is used.

lParam is not used.

Return Value The return value is the number of character positions before the first character in the line.

Notes This message will not be processed by single line edit controls.

EM_LINESCROLL

- Purpose* This message scrolls the context of the control by the given number of lines.
- Parameters* *wParam* is not used.
The high-order word of *lParam* contains the number of lines, and the low-order word contains the number of character positions to scroll horizontally.
- Return Value* None.
- Notes* This message will not be processed by single line edit controls.

EM_SCROLL

- Purpose* This message directs the edit control to scroll the contents of its window vertically.
- Parameters* *wParam* is one of the following values:
- | Value | Meaning |
|------------------|--------------------------------------|
| SB_LINEUP | Scroll one line up. |
| SB_LINEDOWN | Scroll one line down. |
| SB_PAGEUP | Scroll one page up. |
| SB_PAGEDOWN | Scroll one page down. |
| SB_THUMBPOSITION | Scroll to thumb position. |
| EM_GETTHUMB | Retrieve the current thumb position. |

lParam is not used.

- Return Value* If *wParam* is EM_GETTHUMB, the control returns the current thumb position.

- Notes* This message will not be processed by single line edit controls.

EM_LINELENGTH

Purpose This message returns the length of a line (in bytes) in the edit control's text buffer.

Parameters *wParam* is the line number of the desired line, where the line number of the first line is 0. If *wParam* is -1, the length of the current line (the line containing the caret) is returned, not including the length of any selected text. If the current selection spans more than one line, the total length of the lines minus the length of the selected text is returned.

lParam is not used.

Return Value EM_LINELENGTH returns the length of the given line (or lines).

EM_REPLACESEL

Purpose This message replaces the current selection with new text.

Parameters *wParam* is not used.

lParam is a far pointer to a null-terminated string of replacement text.

Return Value None.

EM_SETFONT

Purpose This message sets the edit control font.

Parameters *wParam* is the font ID. The font must be a fixed-pitch font.

lParam is not used.

Return Value None.

EM_GETLINE

Purpose This message copies a line from the edit control.

Parameters *wParam* contains the line number of the line in the control, where the line number of the first line is 0.

lParam contains a far pointer to the buffer where the line will be stored. The first word of the buffer specifies the maximum number of bytes to be copied to the buffer. The copied line is not null-terminated.

Return Value The return value is the number of bytes actually copied.

EM_LIMITTEXT

Purpose This message limits the length (in bytes) of the text the user may enter.

Parameters *wParam* specifies the maximum number of bytes that can be entered. If the user attempts to enter more characters, the edit control beeps and does not accept the characters. If *wParam* is 0, no limit is imposed on the size of the text (until no more memory is available).

lParam is not used.

Return Value None.

Notes EM_LIMITTEXT does not affect text set by WM_SETTEXT or the buffer set by EM_SETHANDLE.

EM_UNDO

Purpose This message undoes the last edit to the edit control. When the user modifies the edit control, the last change is stored in an undo buffer, which grows dynamically as required. If insufficient space is available for the buffer, the undo attempt fails and the edit control is unchanged.

Parameters *wParam* and *lParam* are not used.

Return Value The message returns nonzero if the undo operation is successful. It returns zero if it fails.

Notes Implemented only for multiline edit controls.

EM_CANUNDO

Purpose This message determines whether an edit control can respond correctly to an EM_UNDO message.

Parameters *wParam* and *lParam* are not used.

Return Value The return value is nonzero if the edit control can handle the EM_UNDO message correctly. Otherwise, the return value is zero.

Notes Implemented only for multiline edit controls.

EM_FMTLINES

Purpose This message directs the edit control to add or remove the end-of-line character from word-wrapped text lines.

Parameters If *wParam* is nonzero, the characters CR CR LF (0D 0D 0A hexadecimal) are placed at the end of word-wrapped lines. If *wParam* is zero, the end-of-line marks are removed from the text.

lParam is not used.

Return Value EM_FMTLINES returns nonzero if any formatting was done, zero otherwise.

Notes Lines that end with a "hard" return (that is, a carriage return typed by the user), contain just the characters CR LF at the end of the line. These lines are not affected by the EM_FMTLINES message.

Notice that the size of the text changes when this message is processed.

WM_CUT

Purpose This message sends the current selection to the clipboard in CF_TEXT format, then deletes the selection from the control window.

Parameters *wParam* and *lParam* are not used.

Return Value None.

WM_COPY

Purpose This message sends the current selection to the clipboard in CF_TEXT format.

Parameters *wParam* and *lParam* are not used.

Return Value None.

WM_PASTE

Purpose This message inserts the data from the clipboard into the control window at the current cursor position. Data is inserted only if the clipboard contains data in CF_TEXT format.

Parameters *wParam* and *lParam* are not used.

Return Value None.

WM_CLEAR

Purpose This message deletes the current selection.

Parameters *wParam* and *lParam* are not used.

Return Value None.

8.8.3 List Box Messages

LB_ADDSTRING

Purpose This message adds a string to the list box. If the list box is not sorted, the string is appended to the end of the list. Otherwise, it is inserted into the list after sorting.

Parameters *wParam* is not used.

lParam is a long pointer to the null-terminated string to be added.

Return Value The list box returns the index to the string in the list box. LB_ERR is returned if an error occurs; LB_ERRSPACE is returned if insufficient space is available to store the new string.

LB_INSERTSTRING

Purpose This message inserts a string into the list box. No sorting is performed.

Parameters *wParam* is an index to the position to receive the string. If *wParam* is -1, the string is appended to the end of the list.

The *lParam* parameter is a long pointer to the string.

Return Value The return value is the index of the position where the string was inserted. LB_ERR is returned if an error occurs; LB_ERRSPACE is returned if insufficient space is available to store the new string.

LB_DELETESTRING

Purpose This message deletes a string from the list box.

Parameters *wParam* is an index to the string to be deleted.

lParam is not used.

Return Value The return value is a count of the strings remaining in the list. LB_ERR is returned if an error occurs.

LB_SETSEL

Purpose This message sets the selection state of a string.

Parameters *wParam* is a Boolean value specifying how to set the selection. If *wParam* is nonzero, the string is highlighted; if zero, any highlight is removed.

The low-order word of *lParam* is an index that specifies which string to set. If *lParam* is -1, all the strings in the list are selected.

Return Value LB_ERR is returned if an error occurs.

Notes This message should be used only with multiple selection type list boxes.

LB_SETCURSEL

Purpose This message selects a string and scrolls it into view, if necessary. When the new string is selected, the list box removes the highlight from the previously selected string.

Parameters *wParam* is the index of the string to select.

If *lParam* is -1, the list box is set to have no selection.

Return Value LB_ERR is returned if an error occurs.

Notes This message should only be used with single selection type list boxes.

LB_GETSEL

Purpose This message returns the selection state of an item.

Parameters *wParam* is an index to the item.

lParam is not used.

Return Value LB_GETSEL returns a positive number if the item is selected, zero if not. LB_ERR is returned if an error occurs.

LB_GETCURSEL

Purpose This message returns the index of the currently selected item, if any.

Parameters *wParam* and *lParam* are not used.

Return Value LB_GETCURSEL returns the index of the currently selected item. It returns LB_ERR if no item is selected or the list box has multiple selection type.

LB_GETTEXT

Purpose This message copies a string from the list into a buffer.

Parameters *wParam* is the index of the string to be copied.

lParam is a long pointer to the buffer to receive the string. The buffer must have sufficient space for the string and a terminating null character.

Return Value The return value is the length of the string in bytes, excluding the null character. It is LB_ERR if *wParam* is not a valid index.

LB_GETTEXTLEN

Purpose This message returns the length of a string in the list box.

Parameters *wParam* is an index to the string.

lParam is not used.

Return Value The return value is the length of the string in bytes, not including the terminating null character. LB_ERR is returned if an error occurs.

LB_GETCOUNT

- Purpose* This message returns a count of the number of items in the list box.
- Parameters* *wParam* and *lParam* are not used.
- Return Value* The return value is a count of the number of items in the list box. LB_ERR is returned if an error occurs.

LB_SELECTSTRING

- Purpose* This message changes the current selection to the first string having the specified prefix.
- Parameters* *wParam* is the index of the starting point for the search. The starting point is not included in the search, so the value returned by LB_GETCURSEL can be used as the starting point. If *wParam* is -1, the string is searched from the beginning.
lParam is a long pointer to the prefix string. The string must be null-terminated.
- Return Value* The return value is the index of the selected item. LB_ERR is returned if an error occurs.
- Notes* This message must not be used with list boxes having multiple selection type.
A string is selected only if its initial characters (from the starting point) match the characters in the prefix string.

LB_DIR

- Purpose* This message adds a list of the files from the current directory to the list box. Only files with the attributes specified by the *wParam* parameter and matching the file specification given by the *lParam* parameter are added.
- Parameters* The *wParam* parameter is an MS-DOS attribute value (see DlgDirList for a list of the attributes).
The *lParam* parameter is a long pointer to a file specification string. The string can contain wildcard characters (e.g., *.*).
- Return Value* The return value is a count of items displayed. LB_ERR is returned if an error occurs; LB_ERRSPACE is returned if insufficient space is available to store the new strings.

8.9 Notification Messages

Button and edit controls use the WM_COMMAND message to notify the parent window of actions that occur within the control. Unless an exception is noted, the *wParam* parameter of the WM_COMMAND message contains the control ID, the low-order word of *lParam* contains the edit control window handle, and the high-order word of *lParam* contains a control notification code.

The following sections describe the button, edit control, and list box notification codes.

8.9.1 Button Notification Codes

The following button notification codes apply to buttons with BS_USERBUTTON style only.

Code	Meaning
BN_CLICKED	The button has been clicked.
BN_PAINT	Request to repaint the button.
BN_HILITE	Request to highlight the button.
BN_UNHILITE	Request to remove the highlight.
BN_DISABLE	Request to draw a disabled button.
<i>Note</i>	In each case, the high-order word of the <i>lParam</i> parameter contains a window handle for the button.

8.9.2 Edit Control Notification Codes

Code	Meaning
EN_SETFOCUS	The edit control has obtained the input focus.
EN_KILLFOCUS	The edit control has lost the input focus.
EN_CHANGE	The user has taken some action that may have changed the content of the text.
EN_ERRSPACE	The edit control is out of space.

EN_HSCROLL	The user has clicked the mouse on the edit control's horizontal scroll bar. The parent window is notified before the screen is updated.
EN_VSCROLL	The user has clicked the mouse on the edit control's vertical scroll bar. The parent window is notified before the screen is updated.

8.9.3 List Box Notification Codes

The following notification codes apply only to list box controls that have the LBS_NOTIFY style.

Code	Meaning
LBN_SELCHANGE	The selection has been changed.
LBN_DBLCLK	The user double clicked the mouse button over a string.
LBN_ERRSPACE	Out of memory.

8.10 Scroll Bar Messages

Scroll bar controls send WM_VSCROLL or WM_HSCROLL messages to their parents whenever the user clicks the mouse in the control. The *wParam* parameter contains the same values as defined for the scrolling messages for a standard window. The high-order word of *lParam* contains the window handle of the scroll bar control.

8.11 Non-Client Area Messages

This section describes messages that Windows sends to create and maintain the non-client area of a window. The non-client area is any portion of the window that is outside the client area, such as the caption bar and window frame.

Normally, applications do not process these messages, but send them on to **DefWindowProc** for processing.

WM_NCCREATE

Purpose This message is sent before the WM_CREATE message when a window is first created.

Parameters *wParam* is a handle to the window being created.

lParam is a long pointer to the **CREATESTRUCT** data structure for the window.

Default Action Scroll bars are initialized (the scroll bar data structure is allocated, and the scroll bar position and range are set) and the window text is set. Memory used internally to create and maintain the window is allocated.

Return Value WM_NCCREATE returns a nonzero value if the non-client area is created. If an error occurs, the return value is zero; the **CreateWindow** call will return NULL in this case.

WM_NCDESTROY

Purpose This message is sent after the WM_DESTROY message. It is used to destroy the allocated memory block associated with the window.

Parameters *wParam* and *lParam* are not used.

Default Action Memory associated with the window is freed.

WM_NCCALCSIZE

Purpose This message is sent when the size of a window's client area needs to be calculated.

Parameters *wParam* is not used.

lParam is a long pointer to a **RECT** data structure containing the screen coordinates of the window rectangle (including client area, borders, caption, scroll bars, etc.)

Default Action **DefWindowProc** calculates the size of the client area, based on the window characteristics (presence of scroll bars, menu, etc.) and places the result in the **RECT** data structure.

WM_NCHITTEST

Purpose Every time the mouse is moved, this message is sent to the window containing the mouse cursor (unless a window has captured the mouse).

Parameters *wParam* is not used.

lParam contains the x and y coordinates of the mouse cursor. The x coordinate is in the low-order word; the y-coordinate is in the high-order word. These coordinates are always relative to the top left corner of the window.

Default Action The return value of **DefWindowProc** is one of the following values indicating the position of the mouse cursor:

Value	Position
HTNOWHERE	On the screen background or on a dividing line between windows.
HTERROR	Same as HTNOWHERE except that DefWindowProc produces a system beep to indicate an error.
HTTRANSPARENT	In a window currently covered by another window.
HTCLIENT	Client area.
HTCAPTION	Caption area.
HTSYSMENU	System menu box (close box in child windows).
HTGROWBOX	Size box.
HTMENU	Menu area.
HTHSCROLL	Horizontal scroll bar.
HTVSCROLL	Vertical scroll bar.

Notes The MAKEPOINT macro can be used to convert the *lParam* parameter to a POINT.

WM_NCPAINT

- Purpose* This message is sent to a window when its frame needs painting.
- Parameters* *wParam* and *lParam* are not used.
- Default Action* **DefWindowProc** paints the window frame.
- Notes* An application can intercept this message and paint its own custom window frame. However, remember that the clipping region for a window is always rectangular, even if the shape of the frame is altered.

WM_NCACTIVATE

- Purpose* This message is sent to a window when its caption bar or icon needs to be changed to indicate an active or inactive state.
- Parameters* *wParam* is nonzero if an “active” caption or icon is to be drawn. It is zero for an inactive caption or icon.
- Default Action* **DefWindowProc** draws a gray caption bar for an inactive window, a black caption bar for an active window. An active window that is iconic is framed in white.

WM_NCMOUSEMOVE

- Purpose* This message is sent to a window whenever the mouse is moved in a non-client area of the window.
- Parameters* *wParam* contains the code returned by the WM_NCHITTEST message (see WM_NCHITTEST). *lParam* is a **POINT** data structure containing the x and y screen coordinates of the mouse position.
- Default Action* If appropriate, WM_SYSCOMMAND messages are sent.

WM_NCLBUTTONDOWN

- Purpose* This message is sent to a window whenever the left mouse button is pressed while the mouse is in a non-client area of the window.

Parameters *wParam* contains the code returned by the WM_NCHITTEST message (see WM_NCHITTEST).
lParam is a **POINT** data structure containing the x and y screen coordinates of the mouse position.

Default Action If appropriate, WM_SYSCOMMAND messages are sent.

WM_NCLBUTTONUP

Purpose This message is sent to a window whenever the left mouse button is released in a non-client area of the window.
Parameters *wParam* contains the code returned by the WM_NCHITTEST message (see WM_NCHITTEST).
lParam is a **POINT** data structure containing the x and y screen coordinates of the mouse position.

Default Action If appropriate, WM_SYSCOMMAND messages are sent.

WM_NCLBUTTONDOWNDBLCLK

Purpose This message is sent to a window whenever the left mouse button is double clicked in a non-client area of the window.
Parameters *wParam* contains the code returned by the WM_NCHITTEST message (see WM_NCHITTEST).
lParam is a **POINT** data structure containing the x and y screen coordinates of the mouse position.

Default Action If appropriate, WM_SYSCOMMAND messages are sent.

WM_NCRBUTTONDOWN

Purpose This message is sent to a window whenever the right mouse button is pressed while the mouse is in a non-client area of the window.
Parameters *wParam* contains the code returned by the WM_NCHITTEST message (see WM_NCHITTEST).
lParam is a **POINT** data structure containing the x and y screen coordinates of the mouse position.

Default Action None.

WM_NCRBUTTONUP

Purpose This message is sent to a window whenever the right mouse button is released in a non-client area of the window.

Parameters *wParam* contains the code returned by the WM_NCHITTEST message (see WM_NCHITTEST).

lParam is a **POINT** data structure containing the x and y screen coordinates of the mouse position.

Default Action None.

WM_NCRBUTTONDOWNDBLCLK

Purpose This message is sent to a window whenever the right mouse button is double clicked in a non-client area of the window.

Parameters *wParam* contains the code returned by the WM_NCHITTEST message (see WM_NCHITTEST).

lParam is a **POINT** data structure containing the x and y screen coordinates of the mouse position.

Default Action None.

WM_NCMBUTTONDOWN

Purpose This message is sent to a window whenever the middle mouse button is pressed while the mouse is in a non-client area of the window.

Parameters *wParam* contains the code returned by the WM_NCHITTEST message (see WM_NCHITTEST).

lParam is a **POINT** data structure containing the x and y screen coordinates of the mouse position.

Default Action None.

WM_NCMBUTTONUP

Purpose This message is sent to a window whenever the left mouse button is released in a non-client area of the window.

Parameters *wParam* contains the code returned by the WM_NCHITTEST message (see WM_NCHITTEST).

lParam is a **POINT** data structure containing the x and y screen coordinates of the mouse position.

Default Action None.

WM_NCMBUTTONDOWNDBLCLK

Purpose This message is sent to a window whenever the middle mouse button is double clicked in a non-client area of the window.

Parameters *wParam* contains the code returned by the WM_NCHITTEST message (see WM_NCHITTEST).

lParam is a **POINT** data structure containing the x and y screen coordinates of the mouse position.

Default Action None.

Appendix A

Raster Operation Codes And Definitions

A.1 Introduction 411

A.2 Operation Codes 412

(

(

(

A.1 Introduction

This appendix lists the raster operation codes used by the **PatBlt**, **BitBlt**, and **StretchBlt** functions. The raster operation codes define how GDI combines the bits in a source bitmap with the bits in the destination bitmap.

Each raster operation code represents a Boolean operation in which the source, the currently selected brush, and the destination are combined. The operands used in the operations are:

- S Source bitmap
- P Currently selected brush (also called pattern)
- D Destination bitmap

The Boolean operators used in these operations are:

- o Bitwise Or
- x Bitwise Exclusive Or
- a Bitwise And
- n Bitwise Not (inverse)

All Boolean operations are presented in reverse Polish notation. For example, the operation

PSo

replaces the destination with a combination of the source and brush. The operation

DPSoo

combines the source and brush with the destination. Note that there are alternate spellings of the same function so although a particular spelling may not be in the list, an equivalent form will be.

Each raster operation code is a 32-bit integer value whose high-order word is a Boolean operation index and low-order word is the operation code. The 16-bit operation index is a zero-extended 8-bit value which represents the

result of the Boolean operation on predefined brush, source, and destination values. For example, the operation indexes for the *PSo* and *DPSoo* operations are:

P	S	D		PSo	DPSoo
0	0	0		0	0
0	0	1		0	1
0	1	0		1	1
0	1	1		1	1
1	0	0		1	1
1	0	1		1	1
1	1	0		1	1
1	1	1		1	1

Operation index: 00FC 00FE

In this case, *PSo* has the operation index 00FC (read from bottom up); *DPSoo* has the index 00FE. These values define the location of the corresponding raster operation code in the following table. The *PSo* operation is in line 252 (FC hex) of the table; *DPSoo* in line 254 (hex FE).

The most commonly used raster operations have been given special names in the Windows include file. Programmers should use these names whenever possible in their applications.

A.2 Operation Codes

This section lists the raster operation codes.

Boolean Function In HEX	HEX Rop	Boolean Function In R Polish	Common Name
00	00000042	O	BLACKNESS
01	00010289	DPSoon	-
02	00020C89	DPSona	-
03	000300AA	PSon	-
04	00040C88	SDPona	-
05	000500A9	DPon	-
06	00060865	PDSxnon	-
07	000702C5	PDSaon	-

Raster Operation Codes And Definitions

08	00080F08	SDPnaa	-
09	00090245	PDSxon	-
0A	000A0329	DPna	-
0B	000B0B2A	PSDnaon	-
0C	000C0324	SPna	-
0D	000D0B25	PDSnaon	-
0E	000E08A5	PDSonon	-
0F	000F0001	Pn	-
10	00100C85	PDSona	-
11	001100A6	DSon	NOTSRCCOPY
12	00120868	SDPxnon	-
13	001302C8	SDPaon	-
14	00140869	DPSxnon	-
15	001502C9	DPSaon	-
16	00165CCA	PSDPSanaxx	-
17	00171D54	SSPxDSxaxn	-
18	00180D59	SPxPDxa	-
19	00191CC8	SDPSanaxn	-
1A	001A06C5	PDSPaox	-
1B	001B0768	SDPSxaxn	-
1C	001C06CA	PSDPaox	-
1D	001D0766	DSPDxaxn	-
1E	001E01A5	PDSox	-
1F	001F0385	PDSoan	-
20	00200F09	DPSnaa	-
21	00210248	SDPxon	-
22	00220326	DSna	-
23	00230B24	SPDnaon	-
24	00240D55	SPxDSxa	-
25	00251CC5	PDSPanaxn	-
26	002606C8	SDPSaox	-
27	00271868	SDPSxnox	-
28	00280369	DPSxa	-
29	002916CA	PSDPSaoxxn	-
2A	002A0CC9	DPSana	-
2B	002B1D58	SSPxPDxaxn	-
2C	002C0784	SPDSoax	-
2D	002D060A	PSDnox	-
2E	002E064A	PSDPxoax	-
2F	002FOE2A	PDSnoan	-
30	0030032A	PSna	-
31	00310B28	SDPnaon	-
32	00320688	SDPSoox	-
33	00330008	Sn	-
34	003406C4	SPDSaox	-
35	00351864	SPDSxnox	-
36	003601A8	SDPox	-
37	00370388	SDPoan	-
38	0038078A	PSDPoax	-
39	00390604	SPDnox	-
3A	003A0644	SPDSxox	-

3B	003BOE24	SPDnoan	-
3C	003C004A	PSx	-
3D	003D18A4	SPDSonox	-
3E	003E1B24	SPDSnaox	-
3F	003FOOE4A	PSan	-
40	00400FOA	PSDnaa	-
41	00410249	DPSxon	-
42	00420D5D	SDxDPx	-
43	00431CC4	SPDSanaxn	-
44	00440328	SDna	SRCERASE
45	00450B29	DPSnaon	-
46	004606C6	DSPDaox	-
47	0047076A	PSDPxaxn	-
48	00480368	SDPxax	-
49	004916C5	PDSPDaoxxn	-
4A	004A0789	DPSDoax	-
4B	004B0605	PDSnox	-
4C	004COCC8	SDPana	-
4D	004D1954	SSPxDSxoxxn	-
4E	004E0645	PDSPxox	-
4F	004FOE25	PDSnoan	-
50	00500325	PDna	-
51	00510B26	DSPnaon	-
52	005206C9	DPSDaox	-
53	00530764	SPDSxaxn	-
54	005408A9	DPSonon	-
55	00550009	Dn	-
56	005601A9	DPSox	-
57	00570389	DPSoan	-
58	00580785	PDSPoax	-
59	00590609	DPSnox	-
5A	005A0049	DPx	PATINVERT
5B	005B18A9	DPSDonox	-
5C	005C0649	DPSDxoax	-
5D	005DOE29	DPSnoan	-
5E	005E1B29	DPSDnaox	-
5F	005FOOE9	DPan	-
60	00600365	PDSxa	-
61	006116C6	DSPDSaoxxn	-
62	00620786	DSPDoax	-
63	00630608	SDPnox	-
64	00640788	SDPSoax	-
65	00650606	DSPnox	-
66	00660046	DSx	SRCINVERT
67	006718A8	SPDSonox	-
68	006858A6	DSPDSonoxxn	-
69	00690145	PDSxxn	-
6A	006A01E9	DPSax	-
6B	006B178A	PSDPSoaxxn	-
6C	006C01E8	SDPxax	-
6D	006D1785	PDSPDoaxxn	-

Raster Operation Codes And Definitions

6E	006E1E28	SDPSnoax	-
6F	006FOC65	PDSxnan	-
70	00700CC5	PDSana	-
71	00711D5C	SSDxDaxxn	-
72	00720648	SDPSxox	-
73	00730E28	SDPnoan	-
74	00740646	DSPDxox	-
75	00750E26	DSPnoan	-
76	00761B28	SDPSnaox	-
77	007700E6	DSan	-
78	007801E5	PDSax	-
79	00791786	DSPDSoaxxn	-
7A	007A1E29	DPSDnoax	-
7B	007BOC68	SDPxnan	-
7C	007C1E24	SPDSnoax	-
7D	007DOC69	DPSxnan	-
7E	007EO955	SPxDxso	-
7F	007F03C9	DPSaan	-
80	008003E9	DPSaa	-
81	00810975	SPxDxon	-
82	00820C49	DPSxna	-
83	00831E04	SPDSnoaxn	-
84	00840C48	SDPxna	-
85	00851E05	PDSPnoaxn	-
86	008617A6	DSPDSoaxx	-
87	008701C5	PDSaxn	-
88	008800C6	DSa	SRCAND
89	00891B08	SDPSnaoxn	-
8A	008AOE06	DSPnoa	-
8B	008B0666	DSPDxoxn	-
8C	008COE08	SDPnoa	-
8D	008D0668	SDPSxoxn	-
8E	008E1D7C	SSDxDaxx	-
8F	008FOCE5	PDSanan	-
90	00900C45	PDSxna	-
91	00911E08	SDPSnoaxn	-
92	009217A9	DPSDpoaxx	-
93	009301C4	SPDaxn	-
94	009417AA	PSDPSoaxx	-
95	009501C9	DPSaxn	-
96	00960169	DPSxx	-
97	0097588A	PSDPSonoxx	-
98	00981888	SDPSonoxn	-
99	00990066	DSxn	-
9A	009A0709	DPSnax	-
9B	009B07A8	SDPSoaxn	-
9C	009C0704	SPDnax	-
9D	009D07A6	DSPDpoaxn	-
9E	009E16E6	DSPDSaoaxx	-
9F	009F0345	PDSxan	-
A0	00A000C9	DPa	-

A1	00A11B05	PDSPnaoxn	-
A2	00A20E09	DPSnoa	-
A3	00A30669	DPSDxoxn	-
A4	00A41885	PDSPonoxn	-
A5	00A50065	PDxn	-
A6	00A60706	DSPnax	-
A7	00A707A5	PDSPoaxn	-
A8	00A803A9	DPSoa	-
A9	00A90189	DPSoxn	-
AA	00AA0029	D	-
AB	00AB0889	DPSono	-
AC	00AC0744	SPDSxax	-
AD	00AD06E9	DPSDaoxn	-
AE	00AE0B06	DSPnao	-
AF	00AF0229	DPno	-
BO	00B00E05	PDSnoa	-
B1	00B10665	PDSPxoxn	-
B2	00B21974	SSPxDSxox	-
B3	00B30CE8	SDPanan	-
B4	00B4070A	PSDnax	-
B5	00B507A9	DPSDoaxn	-
B6	00B616E9	DPSDPaoxx	-
B7	00B70348	SDPzan	-
B8	00B8074A	PSDPxax	-
B9	00B906E6	DSPDaoxn	-
BA	00BA0B09	DPSnao	-
BB	00BB0226	DSno	MERGEPAINT
BC	00BC1CE4	SPDSanax	-
BD	00BD0D7D	SDxDxan	-
BE	00BE0269	DPSxo	-
BF	00BF08C9	DPSano	-
CO	00C000CA	PSa	MERGECOPY
C1	00C11B04	SPDSnaoxn	-
C2	00C21884	SPDSonoxn	-
C3	00C3006A	PSxn	-
C4	00C40E04	SPDnoa	-
C5	00C50664	SPDSxoxn	-
C6	00C60708	SPDnax	-
C7	00C707AA	PSDPoaxn	-
C8	00C803A8	SPDpoa	-
C9	00C90184	SPDoxn	-
CA	00CA0749	SPSDxax	-
CB	00CB06E4	SPDSaoxn	-
CC	00CC0020	S	SRCCOPY
CD	00CD0888	SPDpono	-
CE	00CE0B08	SPDnao	-
CF	00CF0224	SPno	-
DO	00D00E0A	PSDnoa	-
D1	00D1066A	PSDPxoxn	-
D2	00D20705	PDSnax	-
D3	00D307A4	SPDSoaxn	-

Raster Operation Codes And Definitions

D4	00D41D78	SSPxPDxax	-
D5	00D50CE9	DPSanan	-
D6	00D616EA	PSDPSaoxx	-
D7	00D70349	DPSxan	-
D8	00D80745	PDSPxax	-
D9	00D906E8	SDPSaoxn	-
DA	00DA1CE9	DPSDanax	-
DB	00DB0D75	SPxDxan	-
DC	00DC0B04	SPDnao	-
DD	00DD0228	SDno	-
DE	00DE0268	SDPxo	-
DF	00DF08C8	SDPan	-
E0	00E003A5	PDSoa	-
E1	00E10185	PDSoxn	-
E2	00E20746	DSPDxax	-
E3	00E306EA	PSDPaoxn	-
E4	00E40748	SDPSxax	-
E5	00E506E5	PDSPaoxn	-
E6	00E61CE8	SDPSanax	-
E7	00E70D79	SPxPDxan	-
E8	00E81D74	SSPxDSxax	-
E9	00E95CE6	DSPDSanaxxn	-
EA	00EA02E9	DPSao	-
EB	00EB0849	DPSxno	-
EC	00EC02E8	SDPao	-
ED	00ED0848	SDPxno	-
EE	00EE0086	DSo	SRCPAINT
EF	00EF0A08	SDPnoo	-
FO	00F00021	P	PATCOPY
F1	00F10885	PDSono	-
F2	00F20B05	PDSnao	-
F3	00F3022A	PSno	-
F4	00F40B0A	PSDnao	-
F5	00F50225	PDno	-
F6	00F60265	PDSxo	-
F7	00F708C5	PDSano	-
F8	00F802E5	PDSao	-
F9	00F90845	PDSxno	-
FA	00FA0089	DPo	-
FB	00FB0A09	DPSnoo	PATPAINT
FC	00FC008A	PSo	-
FD	00FD0AOA	PSDnoo	-
FE	00FE02A9	DPSoo	-
FF	00FF0062	1	WHITENESS

~

~

~

Appendix B

Virtual Key Code Summary

This appendix describes the Windows virtual key set. The virtual key set consists of 256 virtual keys, numbered from 0 to 255. The keys from 0 to 127 are called the Windows-defined virtual keys. The keys from 128 to 255 are called OEM-defined virtual keys. The Windows-defined virtual keys are divided into three groups: standard, extended, and reserved.

Every OEM adaptation of Windows must be capable of generating:

Keys	Definition
Standard Keys	The Windows-defined standard virtual keys.
Letter Combination Keys	Key combinations consisting of a SHIFT, CONTROL, or CONTROL-SHIFT and a letter key (65 through 90).
Cursor Combination Keys	Key combinations consisting of a SHIFT, CONTROL, or CONTROL-SHIFT and a cursor key (left, right, up, and down).
Function Keys	A minimum of 10 function keys, f1 through f10. It is possible to query the OEM layer to determine the actual number of supported function keys.
Function Combination Keys	Key combinations consisting of a SHIFT, CONTROL, or CONTROL-SHIFT and a function key.

Implementation of all other keys is entirely OEM-dependent.

In the listing below, the unused entries are empty, the extended keys are marked with an asterisk (*), and all others are standard. The extended keys 96 through 111 are numeric key pad digits and operators found on a keypad.

0	1	2	3	4	5	6	7
0	shift	space	0	A	Q	*numpad0	f1
1 *lbtn	control	prior	1	B	R	*numpad1	f2
2 *rbtn	menu	next	2	C	S	*numpad2	f3
3 cancel	*pause	*end	3	D	T	*numpad3	f4
4 *mbtn	capital	home	4	E	U	*numpad4	f5
5		left	5	F	V	*numpad5	f6
6		up	6	G	W	*numpad6	f7
7		right	7	H	X	*numpad7	f8
8 backsp		down	8	I	Y	*numpad8	f9
9 tab		*select	9	J	Z	*numpad9	f10
a		*print		K		*	*f11
b	escape	*execute		L		*	*f12
c *clear				M		*	*f13
d return		ins		N		*	*f14
e		del		O		*	*f15
f		*help				*	*f16

If an OEM has a keyboard that has the following symbols on a single key-cap, then the following are suggested as part of the country-specific OEM-defined virtual key set.

	U.S.	Japan	France	Germany	Spain	Italy	Sweden
BA	;	:	\$ *	< >	;	< >	' *
BB	= +	; +	= +	+ *	= +	+ *	+ ?
BC	,	<	,	?	,	,	,
BD	-	- =	-	-	-	-	-
BE	.	>	;
BF	/ ?	/ ?	:	/	ss ?	< >	? < >
CO	bq ~	@ bq	bq lb	ac gr	gr cr	ugr sec ac gr	
DB	[{	[{) deg	Aum	ac um	agr # Aum	
DC	\	yn	ugr %	Oum	Ntl	ogr @ Oum	
DD] }] }	< >	Uum	Ccd	egr eac Arn	
DE	" "	~	cr um	# ^	' "	igr ~ um cr	
DF		- \					

ss-German sharp s

bq-back quote

lb-British Pound

yn-Japanese Yen

sec-section symbol

deg-degree symbol

ac-acute accent

gr-grave accent

cr-circumflex

cd-cedilla

um-umlaut

rn-ring

tl-tilde

Notes

Windows uses the OEM-supplied **ToAscii** function to convert virtual key codes into the corresponding ASCII value. To make this conversion simple, some virtual keys have the same value as the corresponding ASCII character. For example, the virtual key code for VK_A is equal to 65, the ASCII value of the capital letter "A."

Since Windows can be run using the keyboard interface only, the mouse keys (1, 2, and 4) are not part of the standard set.

(

(

(

Appendix C

Font Files

C.1	Introduction	425
C.2	Font File Formats	425
C.2.1	Raster Font File Format	430
C.2.2	Vector Font File Format	431
C.3	ANSI Character Set	432

(

)

)

C.1 Introduction

The standard FONTS.FON file provided with Windows includes fonts for both 2:1 and 1:1 aspect ratio displays and only the fonts whose aspect ratio matches the display are actually used. There are three sizes of fonts provided in the ANSI character set: 6, 8, and 11 pixels high for 2:1 displays and 12, 16, and 22 pixels high for 1:1 displays. There is only one 1:1 and one 2:1 OEM-dependent font. There are both Terminal and Document faces in each size.

Windows is capable of synthesizing attributes, such as bold, italic, and underlined, so such fonts are not in FONTS.FON. The fonts that do not correspond to the user's display aspect ratio are nevertheless generic raster fonts that can be used for output devices such as bitmap printers.

C.2 Font File Formats

Formats for font files are defined for both raster and vector fonts. These formats may be used by smart text generators in some GDI support modules. The vector formats, in particular, are more frequently used by GDI itself than by support modules.

Both types of files begin with information which is common to both, and then continue with information which differs for each type of file. Font files are stored with a .fnt extension of the form **name.fnt**. The information at the beginning of both types of files is as follows:

Field	Definition
dfVersion	Two bytes specifying the version of the file (currently 256).
dfSize	Four bytes specifying the total file size in bytes.
dfCopyright	Sixty (60) bytes specifying copyright information.
dfType	Two bytes specifying the type of font file. The low-order byte is for exclusive GDI use. If the low-order bit of the word is 0, it is a bitmap (raster) font file. If the low-order bit is 1, it is a vector font file. The second bit is reserved and must be zero. If no bits follow in the file and the bits are located in memory at a fixed address

specified in *dfBitsOffset*, the third bit is set to 1; otherwise, the bit is set to 0. The high bit of the low-order byte is set if the font was realized by a device. The remaining bits in the low-order byte are reserved and set to zero.

The high-order byte is reserved for device use and will always be set to zero for GDI realized standard fonts. Physical fonts with the high bit of the low-order byte set may use this byte to describe themselves. GDI will never inspect the high-order byte.

dfPoints	Two bytes specifying the nominal point size at which this character set looks best.
dfVertRes	Two bytes specifying the nominal vertical resolution (dots per inch) at which this character set was digitized.
dfHorizRes	Two bytes specifying the nominal horizontal resolution (dots per inch) at which this character set was digitized.
dfAscent	Two bytes specifying the distance from the top of a character definition cell to the baseline of the typographical font. It is useful for aligning the baseline of fonts of different heights.
dfInternalLeading	Two bytes specifying the amount of leading inside the bounds set by <i>dfPixHeight</i> . Accent marks may occur in this area. This may be zero at the designer's option.
dfExternalLeading	Two bytes specifying the amount of extra leading that the designer requests the application add between rows. Since this area is outside of the font proper, it contains no marks, and will not be altered by text output calls in either OPAQUE or TRANSPARENT mode. This may be zero at the designer's option.
dfItalic	One byte specifying whether the character definition data represent an italic font. The low-order bit is one if the flag is set. All other bits are zero.

dfUnderline	One byte specifying whether the character definition data represent an underlined font. The low-order bit is one if the flag is set. All other bits are zero.
dfStrikeOut	One byte specifying whether the character definition data represent a strikeout font. The low-order bit is one if the flag is set. All other bits are zero.
dfWeight	Two bytes specifying the weight of the characters in the character definition data, on a scale from 1-1000. A weight of 100 is the usual for light type, 200 specifies regular, 300 semibold, and so forth.
dfCharSet	One byte specifying the character set defined by this font. The IBM® PC hardware font has been assigned the designation 377 octal (FF hexadecimal or 255 decimal).
dfPixWidth	Two bytes. For vector fonts, specifies the width of the grid on which the font was digitized. For raster fonts, if <i>dfPixWidth</i> is nonzero, it represents the width for all characters in the bitmap; if it is zero, the font has variable width characters whose widths are specified in the <i>dfCharOffset</i> array.
dfPixHeight	Two bytes specifying the height of the character bitmap (raster fonts), or the height of the grid on which a vector font was digitized.
dfPitchAndFamily	Specifies the pitch and font family. The low bit is set if the font is variable-pitch. The four high-order bits give the family name of the font. The values are:
	FF_DONTCARE (0<<4) FF_ROMAN (1<<4) FF_SWISS (2<<4) FF_MODERN (3<<4) FF_SCRIPT (4<<4) FF_DECORATIVE (5<<4)
	Font families describe in a general way the look of a font. They are intended for specifying fonts when the exact facename desired is not available. The families are:

	FF_DONTCARE Don't care or don't know.
	FF_ROMAN Fonts with variable stroke width, serifed. Times Roman, Century Schoolbook, etc.
	FF_SWISS Fonts with variable stroke width, sans-serifed. Helvetica, Swiss, etc.
	FF_MODERN Fonts with constant stroke width, serifed or sans-serifed. Fixed-pitch fonts are usually modern. Pica, Elite, Courier, etc.
	FF_SCRIPT Cursive, etc.
	FF_DECORATIVE Old English, etc.
dfAvgWidth	Two bytes specifying the width of characters in the font. For fixed-pitch fonts this is the same as <i>dfPixWidth</i> . For variable-pitch fonts this is the width of the character 'X'.
dfMaxWidth	Two bytes specifying the maximum pixel width of any character in the font. For fixed-pitch fonts, this is simply <i>dfPixWidth</i> .
dfFirstChar	One byte specifying the first character code defined by this font. Character definitions are stored only for the characters actually present in a font, so use this field when calculating indexes into either <i>dfBits</i> or <i>dfCharOffset</i> .
dfLastChar	One byte specifying the last character code defined by this font. Note that all characters with codes between <i>dfFirstChar</i> and <i>dfLastChar</i> must be present in the font character definitions.
dfDefaultChar	One byte specifying the character to substitute whenever a string contains a character out of the range <i>dfFirstChar</i> through <i>dfLastChar</i> . The character is given relative to <i>dfFirstChar</i> so that <i>dfDefaultChar</i> is the actual value of the character less <i>dfFirstChar</i> . <i>dfDefaultChar</i> should indicate a special character in the font which is not a space.

dfBreakChar	One byte specifying the character that will define word breaks. This character defines word breaks for word wrapping and word spacing justification. The character is given relative to <i>dfFirstChar</i> so that <i>dfBreakChar</i> is the actual value of the character less <i>dfFirstChar</i> . <i>dfBreakChar</i> is normally $(32 - dfFirstChar)$, which is an ASCII space.
dfWidthBytes	Two bytes specifying the number of bytes in each row of the bitmap (raster fonts). No meaning for vector fonts. <i>dfWidthBytes</i> is always an even quantity so that rows of the bitmap start on word boundaries.
dfDevice	Four bytes specifying the offset in the file to the string giving the device name. For a generic device, this value will be zero (0).
dfFace	Four bytes specifying the offset in the file to the null-terminated string that names the face.
dfBitsPointer	Four bytes specifying the absolute machine address of the bitmap. This is set by GDI at load time. <i>dfBitsPointer</i> is guaranteed to be even.
dfBitsOffset	Four bytes specifying the offset in the file to the beginning of the bitmap information. If the 04h bit in <i>dfType</i> is set, then <i>dfBitsOffset</i> is an absolute address of the bitmap. (Probably in ROM) For raster fonts, it points to a sequence of bytes that makes up the bitmap of the font, whose height is the height of the font, and whose width is the sum of the widths of the characters in the font rounded up to the next word boundary. For vector fonts, it points to a string of bytes or words (depending on the size of the grid on which the font was digitized) specifying the strokes for each character of the font. <i>dfBitsOffset</i> must be even.
dfCharOffset	For proportionally spaced raster fonts, this field contains two bytes giving the offset from the start of each bitmap row for each character in the set. The number of characters present in the set is calculated as $((dfLastChar - dfFirstChar) + 1)$ and one spare, used for the sentinel offset. The total is therefore $((dfLastChar - dfFirstChar) + 2)$

entries, each of two bytes. For equal-width raster fonts, this field collapses to zero size because all widths may be obtained by looking up the width in *dfPixelWidth*.

For fixed-pitch vector fonts, each two-byte entry in this array specifies the offset from the start of the bitmap to the beginning of the string of stroke-specification units for the character. The number of bytes or words to be used for a particular character is calculated in the same fashion as the width of a raster character (i.e., subtract its entry from the next one), so that both types of font require a *sentinel* value at the end of the array of values.

For proportionally spaced vector fonts, each four-byte entry is divided into two two-byte fields. The first field gives the starting offset from the start of the bitmap of the character strokes as for fixed-pitch fonts. The second field gives the pixel width of the character.

<facename>
An ASCII character string specifying the name of the font face. The size of this field is the length of the string plus a null terminator.

<devicename>
An ASCII character string specifying the name of the device if this font file is for a specific device. The size of this field is the length of the string plus a null terminator.

<bitmaps>
This field contains the bitmap definitions. The size of this field depends on the length of the bitmap definitions. Each row of a raster bitmap must start on a word boundary. This implies that the end of each row must be padded to an even length.

C.2.1 Raster Font File Format

In addition to the information in the header to the file, a raster font file contains a string of bytes which is the actual bitmap, just as it will be loaded into contiguous memory. That string begins in the file at the offset specified in the *fiBits* field above.

C.2.2 Vector Font File Format

The header information for a vector font file is as described in Section C.2, "Font File Formats." This section describes some additional information for vector font files.

The *CharOffset* field is used to specify the location and usage of the character strokes in the bitmap area. For fixed-pitch fonts, each two-byte entry is an offset from the start of the bitmap to the beginning of the strokes for the character. For variable-pitch fonts, each four-byte entry consists of two bytes giving the offset (as for fixed-pitch) and two bytes giving the width of the character.

For both fixed- and variable- pitch fonts, the bitmap area is the same. Each character is composed of a series of vectors consisting of a pair of signed relative coordinate pairs starting from the character cell origin. Each pair may be preceded by a special value indicating that the next coordinate is to be a pen-up move. The special pen-up value depends on how the coordinates are stored. For one-byte quantities, it is -128 (080H) and for two-byte quantities, it is -32768 (08000H).

The character cell origin must be at the upper left corner of the cell so that the character hangs down and to the right of where it is placed.

The storage format for the coordinates depends on the size of the font. If either *dfPixHeight* or *dfMaxWidth* is greater than 128, the coordinates are stored as 2-byte quantities; otherwise, they are stored as 1-byte quantities.

C.3 ANSI Character Set

Table C.1**ANSI Character Set For Windows**

	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
00	█	█	█	█	█	█	█	█	█	█	█	█	█	█	█	█
10	█	█	█	"	#	\$	%	&	()	*	+	,	-	:	/
20	!	''	█	█	█	█	█	█	█	█	█	█	█	█	█	?
30	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	0
40	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	^
50	P	Q	R	S	T	U	V	W	X	Y	Z	[]	\	~	-
60	'	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
70	p	q	r	s	t	u	v	w	x	y	z	{	}	—	—	—
80	█	█	█	█	█	█	█	█	█	█	█	█	█	█	█	█
90	█	█	█	█	█	█	█	█	█	█	█	█	█	█	█	█
A0	ı	ç	£	¤	¥	₩	₪	₪	₪	₪	₪	₪	₪	₪	₪	₪
B0	°	±	²	³	²	³	µ	₪	₪	₪	₪	₪	₪	₪	₪	₪
C0	À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Î	Ï	Ñ
D0	Ò	Ó	Ô	Õ	Ô	Õ	Ô	Õ	Ô	Õ	Ô	Õ	Ô	Õ	Ô	Õ
E0	à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	î	ï	ñ
F0	ò	ó	ô	õ	ô	õ	ô	õ	ô	õ	ô	õ	ô	õ	ô	õ

Index

ABORTDOC escape, 176
ABSOLUTE mode, 142
ACCELERATORS resource, 286
Accelerators
 loading, 212
 resource, 286
 translating, 16
AccessResource, 215
Active window, 36
AddAtom, 220
AddFontResource, 207
AllocResource, 214
ALTERNATE mode, 147
ANSI_CHARSET, 263
ANSI_FIXED_FONT object, 128
AnsiLower, 218
AnsiNext, 219
AnsiPrev, 219
AnsiToOem, 217
AnsiUpper, 218
ANSI_VAR_FONT object, 128
AnyPopup, 37
Arc, 114
Area filled, 124
Arg, 342
Ascent, 268
Aspect ratio, 269
ASPECTX capability, 184
ASPECTXY capability, 184
ASPECTY capability, 184
Assembly-language macros. *See*
 Cmacros
assumes macro, 333
Atoms, 219, 220, 221
Attribute functions. *See* Windows,
 attribute functions

Background
 brush, 256
 color
 default, 71
 retrieving, 143
 setting, 142
 erasing, 357

Background (*continued*)
 mode
 default, 71
 retrieving, 144
 setting, 143
Baud rate, 273
BeginPaint, 73
Binary raster operation, 145
Bit level transfers
 copying, 116
 patterns, 115
 stretching, 117
Bit patterns, drawing, 115
BitBlt, 116
BITMAP data structure, 265
Bitmap format, 47, 49, 51. *See also*
 Display format
BITMAP resource, 282
Bitmaps
 bits per pixel, 265
 bits per raster line, 265
 color planes, 265
 compatible with a display, 132
 copying, 116, 117
 creating, 131, 132
 data structure, 265
 deleting, 138
 device-independent format, 209
 height, 265
 loading, 209
 moving, 116
 physical dimensions, 134
 retrieving, 140
 retrieving bits, 133
 selecting, 138
 setting bits, 133
 setting physical dimensions, 133
 stretching, 117
 type, 265
 width, 265
BITSPIXEL capability, 184
BLACK_BRUSH object, 127
BLACKONWHITE mode, 146
BLACK_PEN object, 128
Blink rate (cursor), 320

BM_GETCHECK message, 388
BM_GETSTATE message, 389
BM_SETCHECK message, 389
BM_SETSTATE message, 389
BN_CLICKED code, 401
BN_DISABLE code, 401
BN_HILITE code, 401
BN_PAINT code, 401
BN_UNHILITE code, 401
BOOL type, 251
Border, 28
Break character, 165
BringWindowToTop, 36
Brushes
 color, 260
 creating, 129, 131
 default color, 71
 default origin, 158
 deleting, 138
 enumerating, 181
 hatch style, 261
 hatched, 130
 origin, 158
 patterned, 130
 retrieving, 140
 selecting, 138
 setting origin, 158
 solid, 129
 stock, 127
 style, 260
BS_3STATE style, 302
BS_AUTO3STATE style, 302
BS_AUTOCHECKBOX style, 302
BS_CHECKBOX style, 302
BS_DEFPUSHBUTTON style, 302
BS_GROUPBOX style, 302
BS_HATCHED brush, 260
BS_HOLLOW brush, 260
BS_PATTERN brush, 260
BS_PUSHBUTTON style, 302
BS_RADIOBUTTON style, 302
BS_SOLID brush, 260
BS_USERBUTTON style, 302
BuildCommDCB, 228
BUTTON control class, 300
Button control messages, 388
BUTTON control styles, 302
Button notification codes, 401
BYTE type, 251
Call macros, 341
Calling convention
 Cmacros, 329
 Pascal, 5
CallWindowProc, 25
Capabilities, devices, 183
Caption, 26
Caption bar, 28
CAPTION statement, 294
Capture. *See* Mouse capture
Caret
 creating, 93
 destroying, 93
 displaying, 94
 flash rate, 95
 functions, 92
 hiding, 94
 moving, 94
 position, 94
Carrier detect signal. *See* RLSD signal
Catch, 195
cBegin, 340
cCall, 341
CE_DNS error, 231
CE_FRAME error, 231
CE_IOE error, 231
CE_LOOP error, 231
cEnd, 340
CE_OVERRUN error, 231
CE_PTO error, 231
CE_RXPARITY error, 231
CF_BITMAP format, 49, 51
CF_DIF format, 49, 51
CF_DSPMETAFILEPICT format, 50
CF_DSPTEXT format, 49
CF_GDIOBJFIRST format, 50, 52
CF_GDIOBJLAST format, 50, 52
CF_METAFILEPICT format, 49, 51
CF_OWNERDISPLAY format, 49, 51
CF_PRIVATEFIRST format, 50, 52
CF_PRIVATELAST format, 50, 52
CF_SYLK format, 49, 51
CF_TEXT format, 49, 51
ChangeClipboardChain, 55
ChangeMenu, 62
ChangeMenu flags, 64, 65
char type, 251
Character set, 263, 269
Check box control, 297
CHECKBOX statement, 297
CheckDlgButton, 45

CheckMenuItem, 65
 CheckRadioButton, 46
 Child windows. *See also* Windows,
 child
 clipping, 29
 containing specified point, 96
 creation function, 27
 enumerating, 38
 ChildWindowFromPoint, 96
 Class. *See* Windows, class
 CLASS statement, 295
 Clear To Send signal, 274, 276
 ClearCommBreak, 233
 Client area, 85
 Client coordinates
 converting to screen coordinates, 95
 defined, 95
 final translation origin, 187
 identifying parent window, 96
 retrieving, 85
 ClientToScreen, 95
 Clipboard
 closing, 47
 counting formats, 53
 emptying, 48
 enumerating formats, 53
 functions, 47
 messages, 381
 opening, 47
 registering formats, 52
 retrieving data, 51
 retrieving format names, 53
 retrieving owner handle, 48
 setting data handles, 48
 viewer chain, 54, 55
 Clipboard formats. *See* Formats
 CLIPCAPS capability, 184
 ClipCursor, 91
 Clipping precision, 263
 Clipping region
 checking contents, 161
 child windows, 29
 combining, 162
 default, 71
 described, 159
 modifying, 159, 160
 retrieving, 159
 selecting, 139
 CloseClipboard, 47
 CloseComm, 226
 CloseMetaFile, 169
 CloseSound, 235
 CloseWindow, 35
 CLRDR function code, 234
 CLRRTS function code, 234
 Cmacros
 Arg, 342
 assumes, 333
 calling convention, 329
 calls, 341
 cBegin, 340
 cCall, 341
 cEnd, 340
 codeOFFSET, 333
 cProc, 337
 createSeg, 331
 dataOFFSET, 333
 defined, 327
 DefX, 343
 errn\$, 346
 errnz, 345
 Errors, 345
 example, 346, 347
 externX, 336
 FarPtr, 344
 functions, 337
 globalX, 335
 include file, 327
 labelX, 336
 LocalX, 339
 memory model option, 328
 overriding types, 346
 parmX, 338
 ?PLM, 329
 RegPtr, 344
 Save, 342
 sBegin, 332
 segments, 331
 segNameOFFSET, 334
 sEnd, 332
 special definitions, 343
 stack checking option, 330
 staticX, 334
 storage allocation, 334
 symbol redefinition, 347
 ?WIN, 330
 Windows prolog/epilog, 330
 CMACROS.INC file, 327
 CODE statement, 313
 codeOFFSET macro, 333
 COLOR_ACTIVECAPTION index,
 102, 256

- Color
 - approximating, 187
 - background, 142, 143
 - flood fill operation, 124
 - planes, 265
 - table
 - default, 71
 - setting, 144
 - RGB values, 244, 267
 - setting in WIN.INI, 321
 - system
 - retrieving, 101
 - WM_SYSCOLORCHANGE message, 386
 - system, setting, 102
 - text, 144
- COLOR_ACTIVECAPTION index, 102, 256
- COLOR_BACKGROUND index, 102, 256
- COLOR_CAPTIONTEXT index, 102, 256
- COLOR_INACTIVECAPTION index, 102, 256
- COLOR_MENU index, 102, 256
- COLOR_MENUTEXT index, 102, 256
- COLORONCOLOR mode, 147
- COLOR_SCROLLBAR index, 102, 256
- COLOR_WINDOW index, 102, 256
- COLOR_WINDOWFRAME index, 102, 256
- COLOR_WINDOWTEXT index, 102, 256
- CombineRgn, 162
- Communication device status. *See* COMSTAT data structure
- Communications
 - baud rate, 273
 - binary mode, 274
 - bits per character, 273
 - break control, 233
 - break state, 233
 - closing, 226
 - CTS signal, 274, 276
 - data structures, 272
 - default configuration, 225
 - delay, 276
 - described, 225
 - device control block, 228, 229
 - device state, 229
 - DSR signal, 274, 276
- Communications (*continued*)
 - DTR signal, 274, 275
 - end-of-data character, 276, 277
 - errors, 229
 - errors reading, 227
 - event mask, 231, 232
 - events, 275, 276
 - extended functions, 234
 - flushing queue, 233
 - null characters, 275
 - opening, 225
 - parity, 273, 274, 275
 - reading, 227
 - RLSD signal, 274, 276
 - RTS signal, 274, 275
 - stop bits, 273
 - transmit characters, 228
 - transmit immediate, 277
 - writing, 227
 - XON/XOFF flow control, 274, 275, 276
- COMPLEXREGION code, 140, 159
- COMSTAT data structure, 276
- Control classes, 300, 301
- Control messages, 388
- CONTROL statement, 299
- Control statements
 - CHECKBOX, 297
 - CONTROL, 299
 - CTEXT, 296
 - DEFPUSHBUTTON, 298
 - EDITTEXT, 298
 - general form, 295
 - GROUPBOX, 297
 - ICON, 299
 - LISTBOX, 297
 - LTEXT, 296
 - PUSHBUTTON, 297
 - RADIOBUTTON, 298
 - RTEXT, 296
- Control styles
 - all classes, 306
 - BUTTON, 302
 - EDIT, 302
 - LISTBOX, 304
 - SCROLLBAR, 305
 - STATIC, 304
- Control, yielding, 13, 206
- Controls
 - check box, 297
 - default push button, 298

- Controls (*continued*)
 edit, 298
 group box, 297
 icon, 299
 input to, 360
 list box, 297
 painting, 358
 push button, 297
 radio button, 298
 size box, 301
 text, 296
 user-defined, 299
- Conversions
 device to logical, 188
 display context, 187
 high-order byte, 242
 high-order word, 243
 integer to long pointer, 243
 integers, 242
 logical to device, 188
 long to POINT, 244
 lower to upper case, 218
 low-order byte, 242
 low-order word, 243
 mapping mode, 148
 OEM to ANSI, 218
 strings, 217
 upper to lower case, 218
 word to long, 243
- Coordinates
 functions, 95
 mapping mode, 148
- CopyMetaFile, 170
- CopyRect, 97
- CountClipboardFormats, 53
- CountVoiceNotes, 241
- cProc macro, 337
- CreateBitmap, 131
- CreateBitmapIndirect, 132
- CreateBrushIndirect, 131
- CreateCaret, 93
- CreateCompatibleBitmap, 132
- CreateCompatibleDC, 108
- CreateDC, 107
- CreateDialog, 39
- CreateEllipticRgn, 164
- CreateEllipticRgnIndirect, 164
- CreateFont, 134
- CreateFontIndirect, 137
- CreateHatchBrush, 130
- CreateIC, 108
- CreateMenu, 62
- CreateMetaFile, 169
- CreatePatternBrush, 130
- CreatePen, 128
- CreatePenIndirect, 129
- CreatePolygonRgn, 164
- CreateRectRgn, 163
- CreateRectRgnIndirect, 164
- createSeg macro, 331
- CreateSolidBrush, 129
- CREATESTRUCT data structure, 258
- CreateWindow, 26
- CS_CLASSDC style, 72, 255
- CS_DBLCLKS style, 254
- CS_HREDRAW style, 254
- CS_KEYCWTWINDOW style, 254
- CS_OEMCHARS style, 254
- CS_OWNDC style, 255
- CS_VREDRAW style, 254
- CTEXT statement, 296
- CTL_BTN code, 358
- CTL_DLG code, 358
- CTL_EDIT code, 358
- CTL_LISTBOX code, 358
- CTL_MSGBOX code, 358
- CTL_SCROLLBAR code, 358
- CTL_STATIC code, 358
- CTS signal, 274, 276
- Current position, 110, 111
- CURSOR resource, 282
- Cursor
 blink rate, 320
 class, 255
 defined, 90
 display count, 92
 displaying, 92
 hiding, 92
 loading, 209
 position, 91, 92
 predefined, 209
 shape, 91
- CURVECAPS capability, 184
- Data formats. *See* Formats
- Data Interchange Format. *See* DIF format
- Data interchange. *See* Clipboard functions
- Data Set Ready signal, 274, 276
- DATA statement, 314

Data structures
BITMAP, 265
COMSTAT, 276
CREATESTRUCT, 258
DCB, 272
GDI, 259
LOGBRUSH, 260
LOGFONT, 261
LOGPEN, 259
METAFILEPICT, 271
MSG, 257
OFSTRUCT, 277
PAINTSTRUCT, 257
POINT, 266
RECT, 266
TEXTMETRIC, 267
WNDCLASS, 254

Data Terminal Ready signal, 274, 275
Data types, naming conventions, 4
dataOFFSET macro, 333
DC. *See* Display context
DCB data structure, 272
DEF file. *See* Module definition file
Default output device, 320
Default push button control, 298
Default window function, 21
DEFAULT_PITCH, 264
DEFAULT_QUALITY, 263
define directive, 308
DEFPUSHBUTTON statement, 298
DefWindowProc, 21
DefX, 343
DeleteAtom, 220
DeleteDC, 109
DeleteMetaFile, 170
DeleteObject, 138
Descent, 268
DESCRIPTION statement, 312
DestroyCaret, 93
DestroyMenu, 62
DestroyWindow, 30
Device capabilities, 183
Device control block. *See* DCB data structure
Device drivers
ABORTDOC escape, 176
aspect ratio, 184
brushes, 184
capabilities, 183
character-stream, PLP 183
clipping capabilities, 184

Device drivers (*continued*)
color bits per pixels, 184
color planes, 184
colors, 184
curve capabilities, 184
display-file 183
DRAFTMODE escape, 177
environment, 186
ENDDOC escape, 173
escape functions, 171
FLUSHOUTPUT escape, 179
fonts, 184
GETCOLORTABLE escape, 178
GETPHYSIZESIZE escape, 178
GETPRINTINGOFFSET escape, 178
GETSCALINGFACTOR escape, 179
height in pixels, 184
information, 180
line capabilities, 185
metafile, VDM 183
NEWFRAME escape, 174
NEXTBAND escape, 174
pens, 184
physical data, 184
physical height, 183
physical width, 183
polygonal capabilities, 185
QUERYESCSUPPORT escape, 179
raster camera 183
raster capabilities, 184
raster display, 183
raster printer 183
SETABORTPROC escape, 175
SETCOLORTABLE escape, 177
STARTDOC escape, 172
supported escape codes, 179
technology, 183
text capabilities, 185
vector plotter, 183
version number, 183
width in pixels, 183
Device mode settings, 386
DEVICE_DEFAULT_FONT object, 128
Dialog box
coordinates, 47
dialog item. *See* Dialog item
functions, 39
initializing, 363
list box

list box (*continued*)
 creating, 41
 selection, 43
 modal
 creating, 40
 destroying, 41
 modeless
 creating, 39
 messages, 40
 units, 295
 Dialog item
 handle, retrieving, 43
 integer
 retrieving, 44
 setting, 43
 messages, 46
 text
 retrieving, 45
 setting, 44
 DIALOG resource
 CAPTION statement, 294
 CLASS statement, 295
 creating, 292
 control statements, 295
 dialog option statements, 293
 MENU statement, 294
 STYLE statement, 293
 DIALOG statement. *See* DIALOG
 resource
 DialogBox, 40
 DIF format, 47, 49, 51
 Directives
 # define, 308
 # elif, 309
 # else, 310
 # endif, 310
 # if, 309
 # ifdef, 308
 # ifndef, 309
 # include, 307
 # undef, 308
 DispatchMessage, 17
 Display context
 attributes, 140
 background color, 142, 143
 background mode, 143, 144
 clipping region, 159
 conversions, 187
 creating, 107
 current position, 111
 default attributes, 141
 Display context (*continued*)
 default characteristics, 71
 deleting, 109
 device capabilities, 183
 device environment, 186
 drawing, 110
 drawing mode, 144, 145
 final translation origin, 187
 information, 180
 information context, 108
 intercharacter spacing, 167
 mapping mode, 148, 151
 memory display, 108
 polygon filling mode, 147, 148
 relabs flag, 142
 releasing, 73
 restoring, 110
 retrieving, 70
 for entire window, 72
 saving, 109
 selected objects, 138
 stock objects, 127
 stretching mode, 146, 147
 text color, 144
 viewport, 153, 154, 155, 156, 158
 window, 151, 152, 153, 154, 155
 Display count, 92
 Display device
 compatible memory displays, 108
 Display format
 bitmap, 49, 51
 metafile picture, 50, 51
 text, 49, 51
 DKGRAY_BRUSH object, 127
 DlgDirList, 41
 DlgDirSelect, 43
 DLG_HASSETSEL code, 360
 DLG_WANTALLKEYS code, 360
 DLG_WANTARROWS code, 360
 DLG_WANTTAB code, 360
 Double click speed, 320
 DPtoLP, 188
 DRAFTMODE escape, 177
 DRAFT_QUALITY, 263
 DrawIcon, 122
 Drawing modes
 default, 71
 list of, 270
 retrieving, 145
 setting, 144
 Drawing, described, 110

DrawMenuBar, 65
DrawText, 119
DRIVERVERSION capability, 183
DSR signal, 274, 276
DS_SYSMODAL, 294
DTR signal, 274, 275
DWORD type, 251
Dynamic linking
 code, 192
 data segments, 193, 194
 described, 191

Edit control
 creating, 298
 messages, 390
 notification codes, 401
EDIT control class, 300
EDIT control styles, 302
EDITTEXT statement, 298
elif directive, 309
Ellipses, drawing, 113
Elliptical arcs, 114
else directive, 310
EM_CANUNDO message, 395
EM_FMTLINES message, 396
EM_GETHANDLE message, 392
EM_GETLINE message, 394
EM_GETLINECOUNT message, 392
EM_GETRECT message, 390
EM_GETSEL message, 390
EM_GETTHUMB message, 393
EM_LIMITTEXT message, 395
EM_LINEINDEX message, 392
EM_LINELENGTH message, 394
EM_LINESCROLL message, 393
EmptyClipboard, 48
EM_REPLACESEL message, 394
EM_SCROLL message, 393
EM_SETFONT message, 394
EM_SETHANDLE message, 391
EM_SETRECT message, 391
EM_SETRECTNP message, 391
EM_SETSEL message, 390
EM_UNDO message, 395
EnableMenuItem, 66
EnableWindow, 60
EN_CHANGE code, 401
EndDialog, 41
ENDDOC escape, 173
endif directive, 310

End-of-data character, 276, 277
EndPaint, 74
EN_ERRSPACE code, 401
English units, 149
EN_HSCROLL code, 402
EN_KILLFOCUS code, 401
EN_SETFOCUS code, 401
Entry point, main function, 9
EnumChildWindows, 38
EnumClipboardFormats, 53
EnumFonts, 180
EnumObjects, 181
EnumProps, 83
EnumWindows, 37
EN_VSCROLL code, 402
EqualRgn, 163
errn\$, 346
errnz, 345
ERROR code, 140, 159
Error functions, 87
Errors, 225
ES_AUTOHSCROLL style, 303
ES_AUTOVSCROLL style, 303
Escape, 171, 172, 173, 174, 175, 176,
 177, 178, 179
EscapeCommFunction, 234
Escapement, 262
ES_CENTER style, 302
ES_LEFT style, 302
ES_MULTILINE style, 303
ES_NOHIDESEL style, 303
ES_RIGHT style, 302
EV_BREAK event, 231, 232
EV_CTS event, 231, 232
EV_DSR event, 231, 232
EVENPARITY, 273
EV_ERR event, 231, 232
EV_PERR event, 231
EV_RING event, 231
EV_RLSD event, 231, 232
EV_RXCHAR event, 231, 232
EV_RXFLAG event, 231, 232
EV_TXEMPTY event, 231, 232
ExcludeClipRect, 160
Execution, terminating, 10
EXPORTS statement, 316
Extents, 71
externX macro, 336

Face name, 264

Far pointer, 6
 FAR type, 252
 FARPROC type, 252
 FarPtr, 344
 FatalExit, 225
 FF_DECORATIVE family, 264, 427
 FF_DONTCARE family, 264, 427
 FF_MODERN family, 264, 427
 FF_ROMAN family, 264, 427
 FF_SCRIPT family, 264, 427
 FF_SWISS family, 264, 427
 File attributes, 42
 File formats
 font files, 425
 module definition file, 311
 raster fonts, 430
 resource file, 281
 vector fonts, 431
 Files
 described, 245
 initialization, 222
 opening, 245
 temporary, 246
 temporary drive, 248
 FillRect, 126
 FillRgn, 125
 Final translation origin, 187
 FindAtom, 221
 FindResource, 212
 FindWindow, 85
 FIXED_PITCH, 264
 Flash rate. *See* Caret, flash rate
 Flash Window, 90
 FloodFill, 124
 FlushComm, 233
 FLUSHOUTPUT escape, 179
 Focus. *See* Input, focus
 Font files, 425
 FONT resource, 282
 Fonts
 adding, 207, 386
 creating, 134, 137
 default, 71
 deleting, 138
 enumerating, 180
 logical, 261, 262, 263, 264
 physical, 267, 268, 269
 reference count, 208
 removing, 208, 386
 retrieving, 140
 selecting, 138

Fonts (*continued*)
 stock, 127
 text color, 144
 text metrics, 183
 WIN.INI, 323
 Formats, 47, 52, 53
 FrameRect, 126
 FrameRgn, 125
 FreeLibrary, 194
 FreeProcAddress, 194
 FreeResource, 215
 Function macros, 337
 Functions
 naming conventions, 4
 notational conventions, 3
 GCL_MENU index, 23, 24
 GCL_WNDPROC index, 23, 24
 GCW_CBCLSEXTRA index, 23, 24
 GCW_CBWNDEXTRA index, 23, 24
 GCW_HBRBACKGROUND index, 23, 24
 GCW_HCURSOR index, 23, 24
 GCW_HICON index, 23, 24
 GCW_HINSTANCE index, 23, 24
 GCW_STYLE index, 23, 24
 GDI
 data structures, 259
 functions, 107
 GetAtomName, 221
 GetBitmapBits, 133
 GetBitmapDimension, 134
 GetBkColor, 143
 GetBkMode, 144
 GetBrushOrg, 158
 GetBValue, 244
 GetCaretBlinkTime, 95
 GetClassLong, 23
 GetClassName, 22
 GetClassWord, 23
 GetClientRect, 85
 GetClipboardData, 51
 GetClipboardFormatName, 53
 GetClipboardOwner, 48
 GetClipboardViewer, 54
 GetClipBox, 159
 GetCodeHandle, 193
 GETCOLORTABLE escape, 178
 GetCommError, 229
 GetCommEventMask, 232

Index

GetCommState, 229
GetCurrentPosition, 111
GetCurrentTask, 206
GetCurrentTime, 15
GetCursorPos, 92
GetDC, 70
GetDCOrg, 187
GetDeviceCaps, 183
GetDlgItem, 43
GetDlgItemInt, 44
GetDlgItemText, 45
GetEnvironment, 186
GetFocus, 56
GetGValue, 244
GetInstanceData, 192
GetKeyState, 56
GetMapMode, 151
GetMenu, 62
GetMenuItemString, 69
GetMessage, 10
GetMessagePos, 14
GetMessageTime, 14
GetMetaFile, 169
GetMetaFileBits, 170
GetModuleFileName, 191
GetModuleHandle, 191
GetModuleUsage, 191
GetNearestColor, 187
GetObject, 140
GetParent, 85
GETPHYSPAGESIZE escape, 178
GetPixel, 123
GetPolyFillMode, 148
GETPRINTINGOFFSET escape, 178
GetProcAddress, 192
GetProfileInt, 222
GetProfileString, 223
GetProp, 82
GetRelAbs, 142
GetROP2, 145
GetRValue, 244
GETSCALINGFACTOR escape, 179
GetScrollPos, 79
GetScrollRange, 81
GetStockObject, 127
GetStretchBltMode, 147
GetSubMenu, 68
GetSysColor, 101
GetSysModalWindow, 86
GetSystemMenu, 68
GetSystemMetrics, 100
GetTempDrive, 248
GetTempFileName, 246
GetTextCharacterExtra, 167
GetTextColor, 144
GetTextExtent, 166
GetTextFace, 182
GetTextMetrics, 183
GetThresholdEvent, 241
GetThresholdStatus, 241
GetUpdateRect, 75
GetVersion, 195
GetViewportExt, 158
GetViewportOrg, 154
GetWindowDC, 72
GetWindowExt, 153
GetWindowLong, 31
GetWindowOrg, 151
GetWindowRect, 86
GetWindowText, 84
GetWindowTextLength, 84
GetWindowWord, 30
GlobalAlloc, 196
GlobalCompact, 197
GlobalDiscard, 197
GlobalFlags, 200
GlobalFree, 197
GLOBALHANDLE type, 253
GlobalLock, 198
GlobalReAlloc, 198
GlobalSize, 199
GlobalUnlock, 200
globalX macro, 335
GMEM_DISCARDABLE flag, 196, 199, 200
GMEM_DISCARDED flag, 200
GMEM_FIXED flag, 196, 198
GMEM_LOCKCOUNT flag, 200
GMEM_MODIFY flag, 199
GMEM_MOVEABLE flag, 196, 198
GMEM_NOCOMPACT flag, 196, 199
GMEM_NODISCARD flag, 196, 199
GMEM_SWAPPED flag, 200
GMEM_ZEROINIT flag, 196, 199
Graphic Device Interface, 107. *See also* GDI
Graphs, connected lines, 111
GRAY_BRUSH object, 127
GrayString, 121
Group box control, 297
GROUPBOX statement, 297
GWL_STYLE index, 31, 32

GWL_WNDPROC index, 31, 32
 GWW_HINSTANCE index, 30, 31
 GWW_HWNDPARENT index, 30, 31
 GWW_HWNDTEXT index, 30, 31
 GWW_ID index, 30, 31

HANDLE type, 252
 Handle types, 252
 Hatch style, 261
 HBITMAP type, 253
 HBRUSH type, 253
 HCURSOR type, 253
 HDC type, 253
 Heap
 global
 allocating, 196
 compacting, 197
 condition flags, 200
 described, 196
 discarding, 197
 freeing, 197
 locking, 198
 reallocating, 198
 size, 199
 unlocking, 200
 local
 allocating, 200
 compacting, 201, 202, 203
 condition flags, 206
 data segment, 205
 described, 196
 discarding, 202
 freeing, 202
 freezing, 202
 handle delta, 205
 handle table, 205
 locking, 202
 locking data segment, 205
 melting, 203
 reallocating, 203
 size, 204
 unlocking, 205
 HEAPSIZE statement, 312
 HFONT type, 253
 HIBYTE, 242
 HICON type, 253
 HideCaret, 94
 HIDE_WINDOW flag, 33
 HiliteMenuItem, 67
 HIWORD, 243

HMENU type, 253
 HOLLOW_BRUSH object, 127
 Hook, 103
 HORZRES capability, 183
 HORZSIZE capability, 183
 HPEN type, 253
 HRGN type, 253
 HS_BDIAGONAL hatch style, 261
 HS_CROSS hatch style, 261
 HS_DIAGCROSS hatch style, 261
 HS_FDIAGONAL hatch style, 261
 HS_HORIZONTAL hatch style, 261
 HSTR type, 252
 HS_VERTICAL hatch style, 261
 HTCAPTION code, 404
 HTCLIENT code, 404
 HTERROR code, 404
 HTGROWBOX code, 404
 HTHSCROLL code, 404
 HTMENU code, 404
 HTNOWHERE code, 404
 HTSYSMENU code, 404
 HTTRANSPARENT code, 404
 HTVSCROLL code, 404
 Icon control, 299
 ICON resource, 282
 ICON statement, 299
 Icons
 class, 255
 drawing, 122
 loading, 210
 opening, 34
 predefined, 210
 IDABORT code, 89
 IDCANCEL code, 89
 IDC_ARROW cursor code, 210
 IDC_CROSS cursor code, 210
 IDC_IBEAM cursor code, 210
 IDC_ICON cursor code, 210
 IDC_SIZE cursor code, 210
 IDC_UPARROW cursor code, 210
 IDC_WAIT cursor code, 210
 IDI_APPLICATION icon code, 211
 IDI_ASTERISK icon code, 211
 IDI_EXCLAMATION icon code, 211
 IDIGNORE code, 89
 IDL_HAND icon code, 211
 IDL_QUESTION icon code, 211
 IDNO code, 89

Index

IDOK code, 89
IDRETRY code, 89
IDYES code, 89
IE_BADID error, 226
IE_BAUDRATE error, 226
IE_BYTESIZE error, 226
IE_DEFAULT error, 226
IE_HARDWARE error, 226
IE_MEMORY error, 226
IE_NOOPEN error, 226
IE_OPEN error, 226
if directive, 309
ifdef directive, 308
ifndef directive, 309
IMPORTS statement, 317
include directive, 307
InflateRect, 98
Information context, 108, 109
InitAtomTable, 219
Initialization
 file, 318
 functions, 222, 223, 224
 messages, 362
 strings, 223
Input
 disabling, 60
 enabling, 60, 61
 focus
 acquiring, 56
 described, 55
 identifying window, 56
 messages, 353
 functions, 55
 messages, 364
Instances
 data, 192
 described, 191
 functions, 193, 194
int type, 251
Integer messages, 351
Intercharacter spacing, 71, 167
IntersectClipRect, 159
IntersectRect, 98
InvalidateRect, 75
InvalidateRgn, 76
InvertRect, 127
InvertRgn, 126
IsDialogMessage, 40
IsDlgButtonChecked, 45
IsIconic, 37
IsRectEmpty, 99
IsWindow, 29
IsWindowEnabled, 61
IsWindowVisible, 36
Italic font, 262

Justification, 165

Kerning, 167
Key codes, 419
Key state. *See* Virtual keys, state
KEYLAST code, 12
KillTimer, 60

labelIX macro, 336
LB_ADDSTRING message, 397
LB_DELETESTRING message, 398
LB_DIR message, 400
LB_GETCOUNT message, 400
LB_GETCURSEL message, 399
LB_GETSEL message, 399
LB_GETTEXT message, 399
LB_GETTEXTLEN message, 399
LB_INSERTSTRING message, 397
LBN_DBCLK code, 402
LBN_ERRSPACE code, 402
LBN_SELCCHANGE code, 402
LB_SELECTSTRING message, 400
LB_SETCURSEL message, 398
LB_SETSEL message, 398
LBS_MULTIPLESEL style, 304
LBS_NOREDRAW style, 304
LBS_NOTIFY style, 304
LBS_SORT style, 304
Leading, 268
Libraries, 194
LIBRARY statement, 312
LINECAPS capability, 185
LineDDA, 124
LineTo, 111
Linking
 described, 191
 procedure addresses, 192
 procedure instances, 193, 194
List box
 control, 297
 messages, 397
 notification codes, 402
LISTBOX

- LISTBOX (*continued*)
 control class, 300
 control styles, 304
 statement, 297
- LMEM_DISCARDABLE flag, 201, 204, 206
- LMEM_DISCARDED flag, 206
- LMEM_FIXED flag, 201, 203
- LMEM_LOCKCOUNT flag, 206
- LMEM MODIFY flag, 204
- LMEM_MOVEABLE flag, 201, 203
- LMEM_NOCOMPACT flag, 201, 204
- LMEM_NODISCARD flag, 201, 204
- LMEM_ZEROINIT flag, 201, 204
- LoadAccelerators, 212
- LoadBitmap, 209
- LoadCursor, 209
- LoadIcon, 210
- LoadLibrary, 194
- LoadMenu, 211
- LoadResource, 214
- LoadString, 212
- LOBYTE, 242
- LocalAlloc, 200
- LocalCompact, 201
- LocalDiscard, 202
- LocalFlags, 206
- LocalFree, 202
- LocalFreeze, 202
- LOCALHANDLE type, 253
- LocalHandleDelta, 205
- LocalLock, 202
- LocalMelt, 203
- LocalReAlloc, 203
- LocalSize, 204
- LocalUnlock, 205
- LocalX, 339
- LockData, 205
- LockResource, 214
- LOGBRUSH data structure, 260
- LOGFONT data structure, 261
- LOGPEN data structure, 259
- Long pointer, 6
- LONG type, 251
- LOWORD, 243
- LPINT type, 252
- LPMSG type, 252
- LPRECT type, 252
- LPSTR type, 252
- LPtoDP, 188
- LTEXT statement, 296
- LTGRAY_BRUSH object, 127
- Macros. *See* Cmacros
- Main function, 9
- MAKEINTATOM, 221
- MAKEINTRESOURCE, 243
- MAKELONG, 243
- MAKEPOINT, 244
- MakeProcInstance, 193
- MapDialogRect, 47
- Mapping mode
 default, 71
 retrieving, 151
 setting, 148
 units of measure, 148
- MARKPARITY, 273
- max, 245
- MB_ABORTTRYIGNORE code, 87
- MB_APPLMODAL code, 88
- MB_DEFBUTTON1 code, 88
- MB_DEFBUTTON2 code, 88
- MB_DEFBUTTON3 code, 88
- MB_ICONASTERISK code, 88
- MB_ICONEXCLAMATION code, 88
- MB_ICONHAND code, 88
- MBICONQUESTION code, 88
- MB_OK code, 87
- MB_OKCANCEL code, 87
- MB_RETRYCANCEL code, 87
- MB_SYSTEMMODAL code, 88
- MB_YESNO code, 87
- MB_YESNOCANCEL code, 88
- Memory display, 108
- Memory
 compacting, 196, 197
 described, 196
 discardable, 196
 discarding, 197
 fixed, 196
 global
 allocation, 196
 described, 196
 flags, 200
 freeing, 197
 locking, 198
 reallocation, 198
 unlocking, 200
 heaps, 196
 initializing, 196
 local

- local (*continued*)
 - allocating, 200
 - compacting, 201, 202, 203
 - condition flags, 206
 - data segment, 205
 - described, 196
 - discarding, 202
 - freeing, 202
 - freezing, 202
 - handle delta, 205
 - handle table, 205
 - locking, 202
 - locking data segment, 205
 - melting, 203
 - reallocating, 203
 - size, 204
 - unlocking, 205
 - movable, 196
 - reference count, 198, 200, 202, 205, 206
 - size, 199
- MENU, 294
- Menu bar menu. *See* Top-level menu
- MENU resource, 287
- MENU statement, 294
- Menu
 - bar, redrawing, 65
 - changing, 62
 - class, 256
 - creating, 62
 - destroying, 62
 - functions, 61
 - initializing, 362, 363
 - item
 - checking, 65
 - disabling, 66
 - enabling, 66
 - graying, 66
 - highlighting, 67
 - label, retrieving, 69
 - loading, 211
 - popup
 - defined, 61
 - retrieving handle, 68
 - retrieving handle, 62
 - setting, 61
 - system, 68
 - top-level, 61
 - window, 258
- MENUTITEM SEPARATOR statement, 291
- MENUTITEM statement, 288, 289
- Message box
 - beep, 89
 - creating, 87
- MessageBeep, 89
- MessageBox, 87
- Messages
 - accelerator, 16
 - BM_GETCHECK, 388
 - BM_GETSTATE, 389
 - BM_SETCHECK, 389
 - BM_SETSTATE, 389
 - checking for, 12
 - clipboard, 381
 - contents, 351
 - control
 - button, 388
 - edit, 390
 - list box, 397
 - default processing, 21
 - dispatching, 17
- EM_CANUNDO, 395
- EM_FMTLINES, 396
- EM_GETHANDLE, 392
- EM_GETLINE, 394
- EM_GETLINECOUNT, 392
- EM_GETRECT, 390
- EM_GETSEL, 390
- EM_LIMITTEXT, 395
- EM_LINEINDEX, 392
- EM_LINELENGTH, 394
- EM_LINESCROLL, 393
- EM_REPLACESEL, 394
- EM_SCROLL, 393
- EM_SETFONT, 394
- EM_SETHANDLE, 391
- EM_SETRECT, 391
- EM_SETRECTNP, 391
- EM_SETSEL, 390
- EM_UNDO, 395
- functions, 10
- initialization, 362
- input, 364
- integer, 351
- LB_ADDSTRING, 397
- LB_DELETESTRING, 398
- LB_DIR, 400
- LB_GETCOUNT, 400
- LB_GETCURSEL, 399
- LB_GETSEL, 399
- LB_GETTEXT, 399

Messages (continued)

LB_GETTEXTLEN, 399
 LB_INSERTSTRING, 397
 LB_SELECTSTRING, 400
 LB_SETCURSEL, 398
 LB_SETSEL, 398
 mouse position, 14
 non-client area, 402
 notification, 401
 posting, 10, 18, 19
 processing, 20
 ranges, 351
 registering, 19
 replying to, 19
 reserved, 351
 retrieving, 10
 scroll bar, 402
 sending, 17, 18
 string, 351
 system, 377
 system information, 386
 time of, 14
 translating, 15, 16
 virtual key, 15
 waiting for, 13
 window management, 352
WM_ACTIVATE, 353
WM_ACTIVATEAPP, 354
WMASKCBFORMATNAME, 385
WMCHANGECBCHAIN, 382
WMCHAR, 372
WMCLEAR, 397
WMCLOSE, 361
WMCOMMAND, 374
WMCOPY, 396
WMCREATE, 352
WMCTLCOLOR, 358
WMCUT, 396
WMDEADCHAR, 373
WMDESTROY, 361
WMDESTROYCLIPBOARD, 382
WMDEVMODECHANGE, 386
WMDRAWCLIPBOARD, 382
WMENABLE, 353
WMENDSESSION, 362
WMERASEBKND, 357
WMFONTCCHANGE, 386
WMGETDLGCODE, 360
WMGETTEXT, 359
WMGETTEXTLENGTH, 359
WMHSCROLL, 376

Messages (continued)

WMHSCROLLCLIPBOARD, 385
WMINITDIALOG, 363
WMINITMENU, 363
WMINITMENUPOPUP, 362
WMKEYDOWN, 370
WMKEYUP, 371
WMKILLFOCUS, 353
WMLBUTTONDBLCLK, 368
WMLBUTTONDOWN, 365
WMLBUTTONUP, 365
WMMBUTTONDBLCLK, 370
WMMBUTTONDOWN, 367
WMMBUTTONUP, 368
WMMOUSEMOVE, 364
WMMOVE, 357
WMNCACTIVATE, 405
WMNCCALCSIZE, 403
WMNCCREATE, 403
WMNCDESTROY, 403
WMNCHITTEST, 404
WMNCLBUTTONDBLCLK, 406
WMNCLBUTTONDOWN, 405
WMNCLBUTTONUP, 406
WMNCMBUTTONDBLCLK, 408
WMNCMBUTTONDOWN, 407
WMNCMBUTTONUP, 407
WMNCMOUSEMOVE, 405
WMNPAIN, 405
WMNCRBUTTONDBLCLK, 407
WMNCRBUTTONDOWN, 406
WMNCRBUTTONUP, 407
WMPAINT, 357
WMPAINTCLIPBOARD, 383
WMPASTE, 397
WMQUERYENDSESSION, 361
WMQUERYOPEN, 352
WMQUIT, 362
WMRBUTTONDBLCLK, 369
WMRBUTTONDOWN, 366
WMRBUTTONUP, 366
WMRENDERALLFORMATS, 382
WMRENDERFORMAT, 381
WMSETFOCUS, 353
WMSETREDRAW, 359
WMSETTEXT, 359
WMSETVISIBLE, 352
WMSHOWWINDOW, 355
WMSIZE, 356
WMSIZECLIPBOARD, 383
WMSSYSCCHAR, 379

- Messages (*continued*)
 WM_SYSCOLORCHANGE, 386
 WM_SYSCOMMAND, 380
 WM_SYSDEADCHAR, 380
 WM_SYSKEYDOWN, 377
 WM_SYSKEYUP, 378
 WM_SYSTEMERROR, 387
 WM_TIMECHANGE, 387
 WM_TIMER, 374
 WM_USER, 351
 WM_VSCROLL, 375
 WM_VSCROLLCLIPBOARD, 384
 WM_WININICHANGE, 387
- Metafile picture display format. *See* Display format
- Metafile picture format, 47, 49, 51, 271
- METAFILEPICT data structure, 271
- Metafiles
 closing, 169
 copying, 170
 creating, 169
 deleting, 170
 described, 168
 GDI restrictions, 168
 playing, 170
 retrieving, 169
 retrieving bits, 170
 setting bits, 171
- Metric units, 149
- MF_APPEND menu flag, 64
- MF_BITMAP menu flag, 64
- MF_BYCOMMAND menu flag, 64, 66, 67, 70
- MF_BYPOSITION menu flag, 64, 66, 67, 70
- MF_CHANGE menu flag, 64
- MF_CHECKED menu flag, 64, 66
- MF_DELETE menu flag, 64
- MF_DISABLED menu flag, 64, 66
- MF_ENABLED menu flag, 64, 66
- MF_GRAYED menu flag, 64, 66
- MF_HILITE menu flag, 67
- MF_INSERT menu flag, 64
- MF_MENUBARBREAK menu flag, 64
- MF_MENUBREAK menu flag, 64
- MF_POPUP menu flag, 65
- MF_SEPARATOR menu flag, 64
- MF_STRING menu flag, 65
- MF_UNCHECKED menu flag, 64, 66
- MF_UNHILITE menu flag, 67
- min, 244
- MK_CONTROL, 364, 365, 366, 367, 368, 369, 370
- MK_LBUTTON, 364, 366, 367, 368, 369, 370
- MK_MBUTTON, 364, 365, 366, 367, 368, 369, 370
- MK_RBUTTON, 364, 365, 367, 368, 369, 370
- MK_SHIFT, 364, 365, 366, 367, 368, 369, 370
- MM_LOENGLISH mode, 149
- MM_ANISOTROPIC mode, 150
- MM_HIENGLISH mode, 149
- MM_HIMETRIC mode, 149
- MM_ISOTROPIC mode, 150
- MM_LOMETRIC mode, 149
- MM_TEXT mode, 149
- MM_TWIPS mode, 149
- Modal dialog box. *See* Dialog box
- Modeless dialog box. *See* Dialog box
- Module definition file
 CODE statement, 313
 DATA statement, 314
 DESCRIPTION statement, 312
 EXPORTS statement, 316
 HEAPSIZE statement, 312
 IMPORTS statement, 317
 LIBRARY statement, 312
 NAME statement, 311
 SEGMENT statement, 315
 STACKSIZE statement, 313
 STUB statement, 318
- Modules 191, 194
- Mouse capture, 55, 58
- Mouse cursor. *See* Cursor
- Mouse input, 58, 59
- MoveTo, 110
- MoveWindow, 35
- MS-DOS
 error codes, 194
 file attributes, 42
- MSG data structure, 257
- MSGF_DIALOGBOX code, 104
- MSGF_MENU code, 104
- MSGF_MESSAGEBOX code, 104
- NAME statement, 311
- Naming conventions, 4
- Near pointer, 6
- NEAR type, 252

NEWFRAME escape, 174
 NEXTBAND escape, 174
 Non-client area messages, 402
 NOPARITY, 273
 NORMAL voice, 236
 Notational conventions, 3
 Notification codes, 401, 402
 Null port name, 320
 NULL_BRUSH object, 128
 NULL_PEN object, 128
 NULLREGION code, 140, 159
 NUMBRUSHES capability, 184
 NUMCOLORS capability, 184
 NUMFONTS capability, 184
 NUMPENS capability, 184

Objects
 deleting, 138
 described, 127
 enumerating, 181
 retrieving, 140
 selecting, 138
 stock, 127
 ODDPARITY, 273
 OEM_CHARSET, 263
 OEM_FIXED_FONT object, 128
 OemToAnsi, 218
 OF_CANCEL file style, 246
 OF_CREATE file style, 245
 OF_DELETE file style, 246
 OF_EXIST file style, 245
 OffsetClipRgn, 160
 OffsetRect, 99
 OffsetRgn, 163
 OffsetViewportOrg, 156
 OffsetWindowOrg, 155
 OF_PROMPT file style, 246
 OF_READ file style, 245
 OF_READWRITE file style, 245
 OF_REOPEN file style, 245
 OFSTRUCT data structure, 245, 277
 OF_VERIFY file style, 246
 OF_WRITE file style, 245
 ONE5STOPBITS, 273
 ONESTOPBIT, 273
 OPAQUE mode, 142, 143
 Open file data structure, 277
 OpenClipboard, 47
 OpenComm, 225
 OpenFile, 245

OpenIcon, 34
 OpenSound, 235
 Origin
 brush, 71
 viewport, 71
 window, 71
 Output devices, 320, 322
 Output precision, 263
 Output quality, 263
 Overhang, 269
 Owner display format, 49, 51

Painting
 completing, 74
 functions, 70
 invalidating
 rectangle, 75
 region, 76
 preparation for, 73
 validating
 rectangle, 76
 region, 77
 PaintRgn, 126
 PAINTSTRUCT data structure, 257
 Parent window. *See* Windows, parent
 Parity, 273, 274, 275
 parmX macro, 338
 Pascal calling convention, 5
 PatBlt, 115
 PDEVICESIZE capability, 184
 PeekMessage, 12

Pens
 color, 260
 creating, 128, 129
 default, 71
 deleting, 138
 enumerating, 181
 retrieving, 140
 selecting, 138
 stock, 127
 style, 128, 260
 width, 260

Pies, drawing, 115
 PINT type, 252
 Pitch, 264
 Pixels, 123
 PLANES capability, 184
 Planes. *See* Color, planes
 PlayMetaFile, 170
 ?PLM, 329

- POINT data structure, 244, 266
Pointers
 size, 6
 types, 252
POLYGONALCAPS capability, 185
Polygons
 drawing, 111, 113
 filling mode, 71, 113, 147
Polyline, 111
Popup menu. *See* Menu, popup
POPUP statement, 288, 290
Popup windows. *See* Windows, popup
Ports, 321
PostAppMessage, 19
PostMessage, 18
PostQuitMessage, 10
Printing, 173
PROOF_QUALITY, 263
Property list, 81, 82, 83
PSTR type, 252
PtInRect, 99
PtInRegion, 165
PtVisible, 161
Push button control, 297
PUSHBUTTON statement, 297
- QUERYECSUPPORT escape, 179
- Radio button control, 298
RADIOBUTTON statement, 298
Raster fonts, 430
Raster operation codes, 145, 270, 411
RASTERCAPS capability, 184
RC_BANDING capability, 184
RC_BITBLT capability, 184
RC_SCALING capability, 184
ReadComm, 227
Ready To Send signal, 275
Receive Line Signal Detect signal, 276
RECT data structure
 copying, 97
 defined, 266
 empty, 97, 99
 filling, 97
Rectangles
 client, 85
 containing point, 99
 copying, 97
 data structure, 266
- Rectangles (*continued*)
 drawing, 112
 empty, 97, 99
 expanding, 98
 filled, 126
 framed (border), 126
 functions, 97
 intersecting, 98
 invalidating, 75
 inverting (highlighting), 127
 offsetting, 99
 rounded corners, 112
 shrinking, 98
 union, 98
 updating, 75
 validating, 76
 window, 86
RectVisible, 161
Redraw flag, 359
Regions
 checking contents, 165
 comparing, 163
 creating elliptic, 164
 creating polygons, 164
 creating rectangles, 163, 164
 deleting, 138
 described, 162
 filled, 125
 framed (border), 125
 invalidating, 76
 inverting (highlighting), 126
 moving, 163
 painting, 126
 selecting, 138
 validating, 77
RegisterClass, 22
RegisterClipboardFormat, 52
Registering
 formats, 52
 messages, 19
 window class, 22
RegisterWindowMessage, 19
RegPtr, 344
Relabs flag, 71, 142
RELATIVE mode, 142
ReleaseCapture, 59
ReleaseDC, 73
RemoveFontResource, 208
RemoveProp, 82
ReplyMessage, 19
Request To Send signal, 274

- Reserved messages, 351
 RESETDEV function code, 234
Resource directives. *See Directives*
 Resource file, 281
 Resource statements, 282
Resources
 accelerators, 212, 286
 accessing, 215
 allocating space, 214
 bitmaps, 209, 282
 cursors, 209, 282
 DIALOG, 292
 finding, 212
 font, 207, 208, 282
 freeing, 215
 functions, 207
 handler, 216
 icons, 210, 282
 integer ID, 213
 loading, 214
 locking, 214
 menus, 211, 287
 reference count, 214, 215
 size, 216
 strings, 212, 284
 user-defined, 283
RestoreDC, 110
Revision number, 195
RGB color value, 267
RLSD signal, 274, 276
Rops. *See Raster operation codes*
RoundRect, 112
RT_ACCELERATOR resource code, 213
RT_BITMAP resource code, 213
RT_CURSOR resource code, 213
RT_DIALOG resource code, 213
RTEXT statement, 296
RT_FONT resource code, 213
RT_ICON resource code, 213
RT_MENU resource code, 213
RTS signal, 274, 275
RT_STRING resource code, 213

S_ALLTHRESHOLD state, 240
Save, 342
SaveDC, 109
SB_BOTTOM code, 375, 376, 384, 385
SB_CTL code, 79, 80, 81
sBegin macro, 332

SB_ENDSCROLL code, 375, 376, 384, 385
SB_HORZ code, 79, 80, 81
SB_LINEDOWN code, 375, 376, 384, 385, 393
SB_LINEUP code, 375, 376, 384, 385, 393
SB_PAGEDOWN code, 375, 376, 384, 385, 393
SB_PAGEUP code, 375, 376, 384, 385, 393
SBS_BOTTOMALIGN style, 305
SBS_HORZ style, 305
SBS_LEFTALIGN style, 305
SBS_RIGHTALIGN style, 305
SBS_SIZEBOX style, 305
SBS_SIZEBOXBOTTOMRIGHTALIGN style, 306
SBS_SIZEBOXTOPLEFTALIGN style, 306
SBS_TOPALIGN style, 305
SBS_VERT style, 305
SB_THUMBPOSITION code, 375, 376, 384, 385, 393
SB_THUMBRACK code, 375, 376
SB_TOP code, 375, 376, 384, 385
SB_VERT code, 79, 80, 81
ScaleViewportExt, 155
ScaleWindowExt, 154
SC_CLOSE command, 69, 380
SC_HSCROLL command, 69, 380
SC_ICON command, 69, 380
SC_KEYMENU command, 69, 380
SC_MOUSEMENU command, 69, 380
SC_MOVE command, 69, 380
SC_NEXTWINDOW command, 69, 380
SC_PREVWINDOW command, 69, 380
Screen coordinates, 95, 96
ScreenToClient, 96
Scroll bar
 adding, 80
 horizontal, 28
 messages, 402
 position, 79
 range, 80, 81
 removing, 80
 vertical, 28
SCROLLBAR control class, 301
SCROLLBAR control styles, 305

Index

Scrolling functions, 77
ScrollWindow, 77
SC_SIZE command, 69, 380
SC_VSCROLL command, 69, 380
SC_ZOOM command, 69, 380
SEGMENT statement, 315
segNameOFFSET, 334
SelectClipRgn, 139
SelectObject, 138
sEnd macro, 332
SendDlgItemMessage, 46
SendMessage, 17
SETABORTPROC escape, 175
SetActiveWindow, 36
SetBitmapBits, 133
SetBitmapDimension, 133
SetBkColor, 142
SetBkMode, 143
SetBrushOrg, 158
SetCapture, 58
SetCaretBlinkTime, 95
SetCaretPos, 94
SetClassLong, 24
SetClassWord, 24
SetClipboardData, 48
SetClipboardViewer, 54
SETCOLORTABLE escape, 177
SetCommBreak, 233
SetCommEventMask, 231
SetCommState, 229
SetCursor, 91
SetCursorPos, 91
SetDlgItemInt, 43
SetDlgItemText, 44
SETDTR function code, 234
SetEnvironment, 186
SetFocus, 56
SetMapMode, 148
SetMenu, 61
SetMetaFileBits, 171
SetPixel, 123
SetPolyFillMode, 147
SetPriority, 207
SetProp, 82
SetRect, 97
SetRectEmpty, 97
SetRelAbs, 142
SetResourceHandler, 216
SetROP2, 144
SETRTS function code, 234
SetScrollPos, 79
SetScrollRange, 80
SetSoundNoise, 238
SetStretchBltMode, 146
SetSysColors, 102
SetSysModalWindow, 86
SetTextCharacterExtra, 167
SetTextColor, 144
SetTextJustification, 165
SetTimer, 59
SetViewportExt, 156
SetViewportOrg, 153
SetVoiceAccent, 236
SetVoiceEnvelope, 237
SetVoiceNote, 238
SetVoiceQueueSize, 235
SetVoiceSound, 239
SetVoiceThreshold, 242
SetWindowExt, 152
SetWindowLong, 32
SetWindowOrg, 151
SetWindowsHook, 103
SetWindowText, 84
SetWindowWord, 31
SETXOFF function code, 234
SETXON function code, 234
Short pointer, 6
short type, 251
ShowCaret, 94
ShowCursor, 92
SHOW_FULLSCREEN code, 33
SHOW_ICONWINDOW code, 33
SHOW_OPENNOACTIVATE code, 33
SHOW_OPENWINDOW code, 33
ShowWindow, 33
SIMPLEREGION code, 140, 159
Size box, 28
Size box controls, 301
SIZEFULLSCREEN code, 356
SIZEICONIC code, 356
SIZENORMAL code, 356
SizeofResource, 216
SIZEZOOMHIDE code, 356
SIZEZOOMSHOW code, 356
S_LEGATO voice, 236
SM_CMETRICS index, 101
SM_CURSORLEVEL index, 101
SM_CXBORDER index, 100
SM_CXCURSOR index, 101
SM_CXDLGFRAME index, 100
SM_CXFULLSCREEN index, 101
SM_CXHSCROLL index, 100

SM_CXHTHUMB index, 101
SM_CXICON index, 101
SM_CXSCREEN index, 100
SM_CXVSCROLL index, 100
SM_CYBORDER index, 100
SM_CYCAPTION index, 100
SM_CYCURSOR index, 101
SM_CYDLGFRAME index, 100
SM_CYFULLSCREEN index, 101
SM_CYHSCROLL index, 100
SM_CYICON index, 101
SM_CYKANJIWINDOW index, 101
SM_CYMENU index, 101
SM_CYSCREEN index, 100
SM_CYVSCROLL index, 100
SM_CYVTHUMB index, 100
SM_DEBUG index, 101
SM_FULLSCREEN index, 101
SM_MOUSEPRESENT index, 101
Sound
 accent, 236
 closing, 235
 device state, 240
 functions, 235
 noise generation, 238
 notes in queue, 241
 opening, 235
 queue threshold, 241
 starting play, 240
 stopping play, 240
 synchronizing voices, 241
 threshold, 241
 voice
 envelope, 237
 frequency, 239
 queue size, 235
 threshold, 242
 wave shape, 237
SPACEPARITY, 273
SP_APPABORT error, 174, 175
S_PERIOD1024 source, 238
S_PERIOD2048 source, 238
S_PERIOD512 source, 238
S_PERIODVOICE source, 238
SP_ERROR error, 174, 175
SP_NOTREPORTED bit, 174, 175
SP_OUTOFDISK error, 174, 175
SP_OUTOFCMEMORY error, 174, 175
SP_USERABORT error, 174, 175
S_QUEUEEMPTY state, 240
SS_BLACKFRAME style, 304
SS_BLACKRECT style, 304
SS_CENTER style, 304
S_SERDCC error, 239
S_SERDDR error, 240
S_SERDFQ error, 240
S_SERDLN error, 239
S_SERDMD error, 237
S_SERDNT error, 239
S_SERDRC error, 237
S_SERDSH error, 237
S_SERDSR error, 238
S_SERDTP error, 237
S_SERDVL error, 237, 240
S_SERMACT error, 235
S_SEROFM error, 235
S_SERQFUL error, 237, 239, 240
SS_GRAYFRAME style, 304
SS_GRAYRECT style, 304
SS_ICON style, 304
SS_LEFT style, 304
SS_RIGHT style, 304
S_STACCATO voice, 236
SS_USERITEM style, 304
SS_WHITEFRAME style, 304
SS_WHITERECT style, 304
Stack checking option, 330
STACKSIZE statement, 313
STARTDOC escape, 172
StartSound, 240
Startup options, 320
STATIC control class, 300
STATIC control styles, 304
staticX macro, 334
S_THRESHOLD state, 240
Stop bits, 273
StopSound, 240
StretchBlt, 117
Stretching mode
 default, 71
 retrieving, 147
 setting, 146
Strikeout font, 262
String messages, 351
Strings
 ANSI to OEM conversion, 217
 conversions, 218
 examining characters, 219
 loading, 212
 lower to upper case, 218
 resource, 284
 translations, 217

Index

- Strings (*continued*)
 upper to lower case, 218
STRINGTABLE resource, 284
STUB statement, 318
STYLE statement, 293
Subclass, 25
S_WHITE1024 source, 238
S_WHITE2048 source, 238
S_WHITE512 source, 238
S_WHITEVOICE source, 238
SW_OTHERUNZOOM code, 355
SW_OTHERZOOM code, 355
SW_PARENTCLOSING code, 355
SW_PARENTOPENING code, 355
SYLK format, 47, 49, 51
Symbolic Link format. *See SYLK*
 format
SyncAllVoices, 241
System
 caret. *See Caret*
 colors. *See Color*
 error message, 387
 information functions, 100
 information messages, 386
 menu box, 28, 68
 messages, 377
 metrics, 100
 modal window, 86
 timer. *See Timer*
 time, 387
SYSTEM_FONT object, 128
- Tasks, 206, 207
TECHNOLOGY capability, 183
Termination, 10
Text color, 71, 144
Text control, 296
Text display format. *See Display*
 format
TEXTCAPS capability, 185
Text
 character spacing, 167
 color, 144
 creating fonts, 134
 drawing, 119
 extent, 166
 facename, 182
 graying, 121
 height, 166
 intercharacter spacing, 167
- Text (*continued*)
 justification, 165
 line length, 166
 metrics, 183
 multiple runs, 166
 width, 166
TEXTMETRIC data structure, 267
TextOut, 119
Throw, 195
Tiled windows. *See Windows, tiled*
Time, 14, 15, 387
Timer
 functions, 55
 killing, 60
 setting, 59
Top-level menu, 61
TranslateAccelerator, 16
TranslateMessage, 15
TransmitCommChar, 228
TRANSPARENT mode, 143
Twips, 149
TWOSTOPBITS, 273
Types. *See also Data structures*
 BOOL, 251
 BYTE, 251
 char, 251
 DWORD, 251
 FAR, 252
 FARPROC, 252
 GLOBALHANDLE, 253
 handle, 252
 HBITMAP, 253
 HBRUSH, 253
 HCURSOR, 253
 HDC, 253
 HFONT, 253
 HICON, 253
 HMENU, 253
 HPEN, 253
 HRGN, 253
 HSTR, 252
 int, 251
 LOCALHANDLE, 253
 long, 251
 LPINT, 252
 LPMMSG, 252
 LPRECT, 252
 LPSTR, 252
 NEAR, 252
 PINT, 252
 pointer, 252

Types. *See also* Data structures
(continued)
 PSTR, 252
 short, 251
 VOID, 251
 WORD, 251
 # undef directive, 308

Underlined font, 262
 UngetCommChar, 228
 UnionRect, 98
 Units of measure, 148
 UnlockData, 205
 UnrealizeObject, 158
 Update rectangle, 75
 UpdateWindow, 74
 User profile. *See* WIN.INI
 User-defined control, 299
 User-defined resource, 283

ValidateRect, 76
 ValidateRgn, 77
 VARIABLE_PITCH, 264
 Vector fonts, 431
 Version number, 195
 VERTRES capability, 184
 VERTSIZE capability, 183
 Viewport
 extents, 155, 156, 158
 moving, 156
 origin, 71, 153, 154
 Virtual keys
 codes, 419
 messages, translating, 15
 state, 56
 listed, 57, 58
 VOID type, 251

WaitMessage, 13
 WaitSoundState, 240
 WHITE_BRUSH object, 127
 WHITEONBLACK mode, 146
 WHITE_PEN object, 128
 WH_KEYBOARD hook, 103
 WH_MSGFILTER hook, 103
 ?WIN, 330
 WINDING mode, 147
 Windows

Windows (*continued*)
 active window, 36
 attribute functions, 84
 border, 28
 bounding rectangle, 86
 bringing to top, 36
 caption
 length, 84
 retrieving, 84
 setting, 84
 changing attributes, 31, 32
 changing size, 35
 child
 creating, 26
 enumerating, 38
 initial position, 27
 size, 27
 class
 functions, 22
 background brush, 256
 class name, 22, 256
 cursor, 255
 data structures, 254
 icon, 255
 menu, 256
 registering, 22
 replacing information, 24
 retrieving information, 23
 style, 254
 client area, 85
 closed, 37
 closing, 35
 creating, 26, 258
 destroying, 30
 disabled, 29
 displaying, 33
 enumerating, 37
 extents, 71, 152, 153, 154
 finding, 85
 flashing, 90
 functions
 class, 22
 creation, 26
 default, 21
 display, 32
 hook, 103
 movement, 32
 painting, 70
 subclassing, 25
 height, 259
 hook, 103

Windows (*continued*)
iconic, 28
initial position, 26
initialization file. *See* WIN.INI
messages. *See* Messages
moving, 155
name, 26, 85, 259
opening, 34
origin, 71, 151
parent, 85, 259
popup, 28
 creating, 26
 initial position, 27
 size, 27
 visibility, 37
position, 259
removing, 33
retrieving information, 30, 31
scrolling, 77
styles, 26, 28, 259
subclass, 25
system modal, 86
text, 359
tiled, 26
updating, 74
visibility, 29, 36
width, 259
WindowFromPoint, 96
WIN.INI
 changing, 244, 387
 colors section, 321
 contents, 319
 devices section, 322
 fonts section, 323
 functions, 222
 integers, 222
 international section, 323
 ports section, 321
 search path, 318
 strings, 223
 windows section, 320
WinMain, 9
WM_ACTIVATE, 353
WM_ACTIVATEAPP, 354
WMASKCBFORMATNAME
 message, 49, 385
WM_CHANGECHAIN, 382
WM_CHAR, 372
WM_CLEAR, 397
WM_CLOSE, 361
WM_COMMAND, 374, 401

WM_COPY, 396
WM_CREATE, 352
WM_CTLCOLOR, 358
WM_CUT, 396
WM_DEADCHAR, 373
WM_DESTROY, 30, 361
WM_DESTROYCLIPBOARD, 382
WM_DEVMODECHANGE, 386
WM_DRAWCLIPBOARD, 382
WM_ENABLE, 353
WM_ENDSESSION, 362
WM_ERASEBKND, 357
WM_FONTCHANGE, 386
WM_GETDLGCODE, 360
WM_GETTEXT, 359
WM_GETTEXTLENGTH, 359
WM_HSCROLL, 376
WM_HSCROLLCLIPBOARD, 49, 385
WM_INITDIALOG, 363
WM_INITMENU, 363
WM_INITMENUPOPUP, 362
WM_KEYDOWN, 370
WM_KEYFIRST, 11, 12
WM_KEYLAST, 11
WM_KEYUP, 371
WM_KILLFOCUS, 353
WM_LBUTTONDOWNDBLCLK, 368
WM_LBUTTONDOWN, 365
WM_LBUTTONUP, 365
WM_MBUTTONDOWNDBLCLK, 370
WM_MBUTTONDOWN, 367
WM_MBUTTONUP, 368
WM_MOUSEFIRST, 11, 12
WM_MOUSELAST, 11, 12
WM_MOUSEMOVE, 364
WM_MOVE, 357
WM_NCACTIVATE, 405
WM_NCCALCSIZE, 403
WM_NCCREATE, 403
WM_NCDESTROY, 403
WM_NCHITTEST, 404
WM_NCLBUTTONDOWNDBLCLK, 406
WM_NCLBUTTONDOWN, 405
WM_NCLBUTTONUP, 406
WM_NCMBUTTONDBLCLK, 408
WM_NCMBUTTONDOWN, 407
WM_NCMBUTTONUP, 407
WM_NCMOUSEMOVE, 405
WM_NCPAINT, 405
WM_NCRBUTTONDOWNDBLCLK, 407
WM_NCRBUTTONDOWN, 406

WM_NCRBUTTONUP, 407
WM_PAINT, 357
WM_PAINTCLIPBOARD, 49, 383
WM_PASTE, 397
WM_QUERYENDSESSION, 361
WM_QUERYOPEN, 352
WM_QUIT, 10, 362
WM_RBUTTONDOWNDBLCLK, 369
WM_RBUTTONDOWN, 366
WM_RBUTTONUP, 366
WM_RENDERALLFORMATS, 382
WM_RENDERFORMAT, 381
WM_SETFOCUS, 353
WM_SETREDRAW, 359
WM_SETTEXT, 359
WM_SETVISIBLE, 352
WM_SHOWWINDOW, 355
WM_SIZE, 356
WM_SIZECLIPBOARD, 49, 383
WM_SYSCHAR, 379
WM_SYSCOLORCHANGE, 386
WM_SYSCOMMAND, 380
WM_SYSDEADCHAR, 380
WM_SYSKEYDOWN, 377
WM_SYSKEYUP, 378
WM_SYSTEMERROR, 387
WM_TIMECHANGE, 387
WM_TIMER, 374
WM_USER, 351
WM_VSCROLL, 375
WM_VSCROLLCLIPBOARD, 49, 384
WM_WINICHANGE, 387
WNDCLASS data structure
 defined, 254
 registering, 22
 replacing information, 24
 retrieving information, 23
WndProc, 20
WORD type, 251
WriteComm, 227
WriteProfileString, 224
WS_BORDER, 28
WS_CAPTION, 28
WS_CHILD, 28
WS_CHILDWINDOW, 29
WS_CLIPCHILDREN, 29
WS_CLIPSIBLINGS, 29
WS_DISABLED, 29
WS_DLGFREAME, 28
WS_GROUP, 306
WS_HSCROLL, 28
WS_ICONIC, 28
WS_POPUP, 28
WS_POPUPWINDOW, 29
WS_SIZEBOX, 28
WS_SYSMENU, 28
WS_TABSTOP, 306
WS_TILED, 28
WS_TILEDWINDOW, 29
WS_VISIBLE, 29
WS_VSCROLL, 28

XON/XOFF flow control, 274, 275, 276

Yield, 206
Yielding, 13, 206

(

)

)