Programmer's Reference, Volume 3: Messages, Structures, and Macros

Microsoft®

WINDOWS™

SOFTWARE DEVELOPMENT KIT

Microsoft® Windows™

Version 3.1

Programmer's Reference

Volume 3: Messages, Structures, and Macros

For the Microsoft Windows Operating System

Microsoft Corporation

Document No. PC28917-0492

Contents

Introduction

This manual, *Microsoft Windows Programmer's Reference, Volume 3*, describes the data types, messages, structures, macros, and printer escapes supported by the Microsoft® Windows™ operating system. In addition, dynamic data exchange (DDE) transactions, File Manager events, raster-operation codes, virtual-key codes, and character tables are presented.

Organization of This Manual

Following are brief descriptions of the chapters and appendixes in this manual:

- Chapter 1, "Data Types," describes the keywords that define the size and meaning of parameter and return values associated with the Windows application programming interface (API).
- Chapter 2, "Messages," describes formatted window messages, through which the Windows operating system communicates with applications, and notification messages, which notify a control's parent window of actions that occur within the control.
- Chapter 3, "Structures," defines the data structures associated with the functions that are part of the Windows API.
- Chapter 4, "Macros," describes the purpose and defines the parameters of macros used to help manipulate data in Windows applications.
- Chapter 5, "Printer Escapes," lists printer escapes for the Windows operating system.
- Chapter 6, "Dynamic Data Exchange Transactions," describes the transactions sent by the Dynamic Data Exchange Management Library (DDEML) to an application's dynamic data exchange (DDE) callback function. The transactions notify the application of DDE activity that affects the application.
- Chapter 7, "File Manager Events and Messages," provides descriptions of the events and menu commands File Manager sends to communicate with a File Manager extension dynamic-link library (DLL). The chapter also describes messages the DLL can send File Manager to retrieve information.
- Chapter 8, "Control Panel Messages," lists the messages Control Panel sends to communicate with a Control Panel DLL.

- Chapter 9, "Common Dialog Box Messages," describes the messages a common dialog box can send to notify applications that the user has made or changed a selection in the dialog box.
- Chapter 10, "Installable Driver Messages," lists the messages the Windows operating system sends to notify installable drivers about specific events.
- Appendix A, "Binary and Ternary Raster-Operation Codes," lists and describes the binary and ternary raster operations used by the graphics device interface (GDI).
- Appendix B, "Virtual-Key Codes," shows the symbolic constant names, hexadecimal values, and keyboard equivalents for Windows virtual-key codes.
- Appendix C, "Character Tables," illustrates the Windows character set, the Symbol character set, and the OEM character set used by the Windows operating system.

Document Conventions

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

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

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

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

Data Types

Chapter 1

The data types in this chapter are keywords that define the size and meaning of parameters and return values associated with functions for the Microsoft Windows operating system, version 3.1. The following table contains character, integer, and Boolean types; pointer types; and handles. The character, integer, and Boolean types are common to most C compilers. Most of the pointer-type names begin with a prefix of P, N (for near pointers), or LP (for long pointers). A near pointer accesses data within the current data segment, and a long pointer contains a 32-bit segment:offset value. A Windows application uses a handle to refer to a resource that has been loaded into memory. Windows provides access to these resources through internally maintained tables that contain individual entries for each handle. Each entry in the handle table contains the address of the resource and a means of identifying the resource type.

The Windows data types are defined in the following table:

Type	Definition
ABORTPROC	32-bit pointer to an **AbortProc** callback function.
ATOM	16-bit value used as an atom handle.
BOOL	16-bit Boolean value.
BYTE	8-bit unsigned integer. Use **LPBYTE** to create 32-bit pointers. Use **PBYTE** to create pointers that match the compiler memory model.
CATCHBUF[9]	18-byte buffer used by the **Catch** function.
COLORREF	32-bit value used as a color value.
DLGPROC	32-bit pointer to a dialog box procedure.
DWORD	32-bit unsigned integer or a segment:offset address. Use **LPDWORD** to create 32-bit pointers. Use **PDWORD** to create pointers that match the compiler memory model.
FARPROC	32-bit pointer to a function.
FNCALLBACK	32-bit value identifying the **DdeCallback** function. Use **PFNCALLBACK** to create pointers that match the compiler memory model.
FONTENUMPROC	32-bit pointer to an **EnumFontsProc** callback function.
GLOBALHANDLE	16-bit value used as a handle to a global memory object.
GNOTIFYPROC	32-bit pointer to a **NotifyProc** callback function.
GOBJENUMPROC	32-bit pointer to a **EnumObjectsProc** callback function.
GRAYSTRINGPROC	32-bit pointer to a **GrayStringProc** callback function.

Type	Definition
HANDLE	16-bit value used as a general handle. Use **LPHANDLE** to create 32-bit pointers. Use **SPHANDLE** to create 16-bit pointers. Use **PHANDLE** to create pointers that match the compiler memory model.
HCURSOR	16-bit value used as a cursor handle.
HFILE	16-bit value used as a file handle.
HGDIOBJ	16-bit value used as a graphics device interface (GDI) object handle.
HGLOBAL	16-bit value used as a handle to a global memory object.
HHOOK	32-bit value used as a hook handle.
HKEY	32-bit value used as a handle to a key in the registration database. Use **PHKEY** to create 32-bit pointers.
HLOCAL	16-bit value used as a handle to a local memory object.
HMODULE	16-bit value used as a module handle.
HOBJECT	16-bit value used as a handle to an OLE object.
HWND	16-bit value used as a handle to a window.
HOOKPROC	32-bit pointer to a hook procedure.
HRSRC	16-bit value used as a resource handle.
LHCLIENTDOC	32-bit value used as a handle to an OLE client document.
LHSERVER	32-bit value used as a handle to an OLE server.
LHSERVERDOC	32-bit value used as a handle to an OLE server document.
LINEDDAPROC	32-bit pointer to a **LineDDAProc** callback function.
LOCALHANDLE	16-bit value used as a handle to a local memory object.
LONG	32-bit signed integer.
LPABC	32-bit pointer to an **ABC** structure.
LPARAM	32-bit signed value passed as a parameter to a window procedure or callback function.
LPBI	32-bit pointer to a **BANDINFOSTRUCT** structure.
LPBITMAP	32-bit pointer to a **BITMAP** structure. Use **NPBITMAP** to create 16-bit pointers. Use **PBITMAP** to create pointers that match the compiler memory model.

Type	Definition
LPBITMAPCOREHEADER	32-bit pointer to a **BITMAPCOREHEADER** structure. Use **PBITMAPCOREHEADER** to create pointers that match the compiler memory model.
LPBITMAPCOREINFO	32-bit pointer to a **BITMAPCOREINFO** structure. Use **PBITMAPCOREINFO** to create pointers that match the compiler memory model.
LPBITMAPFILEHEADER	32-bit pointer to a **BITMAPFILEHEADER** structure. Use **PBITMAPFILEHEADER** to create pointers that match the compiler memory model.
LPBITMAPINFO	32-bit pointer to a **BITMAPINFO** structure. Use **PBITMAPINFO** to create pointers that match the compiler memory model.
LPBITMAPINFOHEADER	32-bit pointer to a **BITMAPINFOHEADER** structure. Use **PBITMAPINFOHEADER** to create pointers that match the compiler memory model.
LPCATCHBUF	32-bit pointer to a **CATCHBUF** array.
LPCBT_CREATEWND	32-bit pointer to a **CBT_CREATEWND** structure.
LPCHOOSECOLOR	32-bit pointer to a **CHOOSECOLOR** structure.
LPCHOOSEFONT	32-bit pointer to a **CHOOSEFONT** structure.
LPCLIENTCREATESTRUCT	32-bit pointer to a **CLIENTCREATESTRUCT** structure.
LPCOMPAREITEMSTRUCT	32-bit pointer to a **COMPAREITEMSTRUCT** structure. Use **PCOMPAREITEMSTRUCT** to create pointers that match the compiler memory model.
LPCPLINFO	32-bit pointer to a **CPLINFO** structure. Use **PCPLINFO** to create pointers that match the compiler memory model.
LPCREATESTRUCT	32-bit pointer to a **CREATESTRUCT** structure.
LPCSTR	32-bit pointer to a nonmodifiable character string.
LPCTLINFO	32-bit pointer to a **CTLINFO** structure. Use **PCTLINFO** to create pointers that match the compiler memory model.
LPCTLSTYLE	32-bit pointer to a **CTLSTYLE** structure. Use **PCTLSTYLE** to create pointers that match the compiler memory model.
LPDCB	32-bit pointer to a **DCB** structure.
LPDEBUGHOOKINFO	32-bit pointer to a **DEBUGHOOKINFO** structure.

Type	Definition
LPDELETEITEMSTRUCT	32-bit pointer to a **DELETEITEMSTRUCT** structure. Use **PDELETEITEMSTRUCT** to create pointers that match the compiler memory model.
LPDEVMODE	32-bit pointer to a **DEVMODE** structure. Use **NPDEVMODE** to create 16-bit pointers. Use **PDEVMODE** to create pointers that match the compiler memory model.
LPDEVNAMES	32-bit pointer to a **DEVNAMES** structure.
LPDOCINFO	32-bit pointer to a **DOCINFO** structure.
LPDRAWITEMSTRUCT	32-bit pointer to a **DRAWITEMSTRUCT** structure. Use **PDRAWITEMSTRUCT** to create pointers that match the compiler memory model.
LPDRIVERINFOSTRUCT	32-bit pointer to a **DRIVERINFOSTRUCT** structure.
LPDRVCONFIGINFO	32-bit pointer to a **DRVCONFIGINFO** structure. Use **PDRVCONFIGINFO** to create pointers that match the compiler memory model.
LPEVENTMSG	32-bit pointer to a **EVENTMSG** structure. Use **NPEVENTMSG** to create 16-bit pointers. Use **PEVENTMSG** to create pointers that match the compiler memory model.
LPDRIVERINFOSTRUCT	32-bit pointer to a **DRIVERINFOSTRUCT** structure.
LPFINDREPLACE	32-bit pointer to a **FINDREPLACE** structure.
LPFMS_GETDRIVEINFO	32-bit pointer to a **FMS_GETDRIVEINFO** structure.
LPFMS_GETFILESEL	32-bit pointer to a **FMS_GETFILESEL** structure.
LPFMS_LOAD	32-bit pointer to a **FMS_LOAD** structure.
LPHANDLETABLE	32-bit pointer to a **HANDLETABLE** structure. Use **PHANDLETABLE** to create pointers that match the compiler memory model.
LPHELPWININFO	32-bit pointer to a **HELPWININFO** structure. Use **PHELPWININFO** to create pointers that match the compiler memory model.
LPINT	32-bit pointer to a 16-bit signed value. Use **PINT** to create pointers that match the compiler memory model.
LPKERNINGPAIR	32-bit pointer to a **KERNINGPAIR** structure.

Type	Definition
LPLOGBRUSH	32-bit pointer to a **LOGBRUSH** structure. Use **NPLOGBRUSH** to create 16-bit pointers. Use **PLOGBRUSH** to create pointers that match the compiler memory model.
LPLOGFONT	32-bit pointer to a **LOGFONT** structure. Use **NPLOGFONT** to create 16-bit pointers. Use **PLOGFONT** to create pointers that match the compiler memory model.
LPLOGPALETTE	32-bit pointer to a **LOGPALETTE** structure. Use **NPLOGPALETTE** to create 16-bit pointers. Use **PLOGPALETTE** to create pointers that match the compiler memory model.
LPLOGPEN	32-bit pointer to a **LOGPEN** structure. Use **NPLOGPEN** to create 16-bit pointers. Use **PLOGPEN** to create pointers that match the compiler memory model.
LPLONG	32-bit pointer to a 32-bit signed integer. Use **PLONG** to create pointers that match the compiler memory model.
LPMAT2	32-bit pointer to a **MAT2** structure.
LPMDICREATESTRUCT	32-bit pointer to an **MDICREATESTRUCT** structure.
LPMEASUREITEMSTRUCT	32-bit pointer to a **MEASUREITEMSTRUCT** structure. Use **PMEASUREITEMSTRUCT** to create pointers that match the compiler memory model.
LPMETAFILEPICT	32-bit pointer to a **METAFILEPICT** structure.
LPMETARECORD	32-bit pointer to a **METARECORD** structure. Use **PMETARECORD** to create pointers that match the compiler memory model.
LPMOUSEHOOKSTRUCT	32-bit pointer to a **MOUSEHOOKSTRUCT** structure.
LPMSG	32-bit pointer to an **MSG** structure. Use **NPMSG** to create 16-bit pointers. Use **PMSG** to create pointers that match the compiler memory model.
LPNCCALCSIZE_PARAMS	32-bit pointer to an **NCCALCSIZE_PARAMS** structure.
LPNEWCPLINFO	32-bit pointer to an **NEWCPLINFO** structure. Use **PNEWCPLINFO** to create pointers that match the compiler memory model.

Type	Definition
LPNEWTEXTMETRIC	32-bit pointer to a **NEWTEXTMETRIC** structure. Use **NPNEWTEXTMETRIC** to create 16-bit pointers. Use **PNEWTEXTMETRIC** to create pointers that match the compiler memory model.
LPOFSTRUCT	32-bit pointer to an **OFSTRUCT** structure. Use **NPOFSTRUCT** to create 16-bit pointers. Use **POFSTRUCT** to create pointers that match the compiler memory model.
LPOLECLIENT	32-bit pointer to **OLECLIENT** structure.
LPOLECLIENTVTBL	32-bit pointer to **OLECLIENTVTBL** structure.
LPOLEOBJECT	32-bit pointer to **OLEOBJECT** structure.
LPOLEOBJECTVTBL	32-bit pointer to **OLEOBJECTVTBL** structure.
LPOLESERVER	32-bit pointer to **OLESERVER** structure.
LPOLESERVERDOC	32-bit pointer to **OLESERVERDOC** structure.
LPOLESERVERDOCVTBL	32-bit pointer to **OLESERVERDOCVTBL** structure.
LPOLESERVERVTBL	32-bit pointer to **OLESERVERVTBL** structure.
LPOLESTREAM	32-bit pointer to **OLESTREAM** structure.
LPOLESTREAMVTBL	32-bit pointer to **OLESTREAMVTBL** structure.
LPOLETARGETDEVICE	32-bit pointer to **OLETARGETDEVICE** structure.
LPOPENFILENAME	32-bit pointer to **OPENFILENAME** structure.
LPOUTLINETEXTMETRIC	32-bit pointer to an **OUTLINETEXTMETRIC** structure.
LPPAINTSTRUCT	32-bit pointer to a **PAINTSTRUCT** structure. Use **NPPAINTSTRUCT** to create 16-bit pointers. Use **PPAINTSTRUCT** to create pointers that match the compiler memory model.
LPPALETTEENTRY	32-bit pointer to a **PALETTEENTRY** structure.
LPPOINT	32-bit pointer to a **POINT** structure. Use **NPPOINT** to create 16-bit pointers. Use **PPOINT** to create pointers that match the compiler memory model.
LPPOINTFX	32-bit pointer to a **POINTFX** structure.
LPPRINTDLG	32-bit pointer to a **PRINTDLG** structure.
LPRASTERIZER_STATUS	32-bit pointer to a **RASTERIZER_STATUS** structure.
LPRECT	32-bit pointer to a **RECT** structure. Use **NPRECT** to create 16-bit pointers. Use **PRECT** to create pointers that match the compiler memory model.

Type	Definition
LPRGBQUAD	32-bit pointer to a **RGBQUAD** structure.
LPRGBTRIPLE	32-bit pointer to a **RGBTRIPLE** structure.
LPSEGINFO	32-bit pointer to a **SEGINFO** structure.
LPSIZE	32-bit pointer to a **SIZE** structure. Use **NPSIZE** to create 16-bit pointers. Use **PSIZE** to create pointers that match the compiler memory model.
LPSTR	32-bit pointer to a character string. Use **NPSTR** to create 16-bit pointers. Use **PSTR** to create pointers that match the compiler memory model.
LPTEXTMETRIC	32-bit pointer to a **TEXTMETRIC** structure. Use **NPTEXTMETRIC** to create 16-bit pointers. Use **PTEXTMETRIC** to create pointers that match the compiler memory model.
LPTTPOLYCURVE	32-bit pointer to a **TTPOLYCURVE** structure.
LPTTPOLYGONHEADER	32-bit pointer to a **TTPOLYGONHEADER** structure.
LPVOID	32-bit pointer to an unspecified type.
LPWINDOWPLACEMENT	32-bit pointer to a **WINDOWPLACEMENT** structure. Use **PWINDOWPLACEMENT** to create pointers that match the compiler memory model.
LPWINDOWPOS	32-bit pointer to a **WINDOWPOS** structure.
LPWNDCLASS	32-bit pointer to a **WNDCLASS** structure. Use **NPWNDCLASS** to create 16-bit pointers. Use **PWNDCLASS** to create pointers that match the compiler memory model.
LPWORD	32-bit pointer to a 16-bit unsigned value. Use **PWORD** to create pointers that match the compiler memory model.
LRESULT	32-bit signed value returned from a window procedure or callback function.
MFENUMPROC	32-bit pointer to an **EnumMetaFileProc** callback function.
NEARPROC	16-bit pointer to a function.
OLECLIPFORMAT	16-bit value used as a standard clipboard format.
PATTERN	Equivalent to the **LOGBRUSH** structure. Use **LPPATTERN** to create 32-bit pointers. Use **NPPATTERN** to create 16-bit pointers. Use **PPATTERN** to create pointers that match the compiler memory model.
PCONVCONTEXT	32-bit pointer to a **CONVCONTEXT** structure.
PCONVINFO	32-bit pointer to a **CONVINFO** structure.

Type	Definition
PHSZPAIR	32-bit pointer to a **HSZPAIR** structure.
PROPENUMPROC	32-bit pointer to an **EnumPropFixedProc** or **EnumPropMovableProc** callback function.
RSRCHDLRPROC	32-bit pointer to a **LoadProc** callback function.
TIMERPROC	32-bit pointer to a **TimerProc** callback function.
UINT	16-bit unsigned value.
WNDENUMPROC	32-bit pointer to an **EnumWindowsProc** callback function.
WNDPROC	32-bit pointer to a window procedure.
WORD	16-bit unsigned value.
WPARAM	16-bit signed value passed as a parameter to a window procedure or callback function.

Messages

Chapter 2

The Microsoft Windows operating system communicates with applications through formatted window messages. These messages are sent to an application's window procedure for processing.

Some messages return values that contain information about the success of the message or contain other data needed by an application. To obtain the return value, the application must call the **SendMessage** function to send the message to a window. This function does not return until the message has been processed.

If the application does not require the return value of the message, it can call the **PostMessage** function to send the message. This function places a message in a window's application queue and then returns immediately. If a message does not have a return value, the application can use either function to send the message, unless the message description indicates otherwise.

A message consists of three parts: a message number, a word parameter, and a long parameter. Message numbers are identified by predefined message names. Each message name begins with letters that suggest the meaning or origin of the message. The word parameter and long parameter, named *wParam* and *lParam* respectively, contain values that depend on the message number.

The *lParam* parameter often contains more than one type of information. For example, the high-order word may contain a handle to a window and the low-order word may contain an integer value. The **HIWORD** and **LOWORD** utility macros can be used to extract the high- and low-order words of the *lParam* parameter. The **HIBYTE** and **LOBYTE** utility macros can be used with **HIWORD** and **LOWORD** to access any of the bytes. Casting can also be used.

Following are the four ranges of message numbers:

Range	Meaning
0 through WM_USER – 1	Messages reserved for use by Windows.
WM_USER through 0x7FFF	Integer messages for use by applications.
0x8000 through 0xBFFF	Messages reserved for use by Windows.
0xC000 through 0xFFFF	String messages for use by applications.

Message numbers in the first range (0 through WM_USER – 1) are defined by Windows. Values in this range that are not explicitly defined are reserved for future use by Windows. This chapter describes messages in this range.

Message numbers in the second range (WM_USER through 0x7FFF) can be defined and used by an application to send messages within a private window class. Such predefined control classes as BUTTON, EDIT, LISTBOX, and COMBOBOX may use values in this range. Messages in this range should not be sent to other applications unless the applications have been designed to exchange messages and to attach the same meaning to the message numbers.

Message numbers in the third range (0x8000 through 0xBFFF) are reserved for future use by Windows.

Message numbers in the fourth range (0xC000 through 0xFFFF) are defined at run time when an application calls the **RegisterWindowMessage** function to obtain a message number for a string. All applications that register the identical string can use the associated message number for exchanging messages with each other. The actual message number, however, is not a constant and cannot be assumed to be the same in different Windows sessions.

2.1 Window Messages

This section describes window messages. These messages are presented in alphabetic order.

BM_GETCHECK

2.x

```
BM_GETCHECK
wParam = 0;      /* not used, must be zero */
lParam = 0L;     /* not used, must be zero */
```

An application sends a BM_GETCHECK message to retrieve the check state of a button.

Parameters

This message has no parameters.

Return Value

The return value from a button created with the BS_AUTOCHECKBOX, BS_AUTORADIOBUTTON, BS_AUTO3STATE, BS_CHECKBOX, BS_RADIOBUTTON, or BS_3STATE style may be one of the following values:

Value	Meaning
0	Button state is unchecked.
1	Button state is checked.
2	Button state is indeterminate (applies only if the button has the BS_3STATE or BS_AUTO3STATE style).

If the button has any other style, the return value is 0.

Example

This example determines if the ID_MYCHECKBOX control is currently checked:

```
int checked;

checked = (int) SendDlgItemMessage(hwndDlg, ID_MYCHECKBOX,
    BM_GETCHECK, 0, 0L);
```

See Also

BM_GETSTATE, BM_SETCHECK

BM_GETSTATE

2.x

```
BM_GETSTATE
wParam = 0;      /* not used, must be zero */
lParam = 0L;     /* not used, must be zero */
```

An application sends a BM_GETSTATE message to retrieve the state of a button.

Parameters

This message has no parameters.

Return Value

The return value specifies the current state of the button. You can use the following masks to extract information about the state:

Mask	Description
0x0003	Specifies the check state (radio buttons and check boxes only). A value of 0 indicates the button is unchecked. A value of 1 indicates the button is checked. A radio button is checked when it contains a dot; a check box is checked when it contains an X. A value of 2 indicates the check state is indeterminate (3-state check boxes only). The state of a 3-state check box is indeterminate when it is grayed.
0x0004	Specifies the highlight state. A nonzero value indicates that the button is highlighted. A button is highlighted when the user presses and holds the left mouse button. The highlighting is removed when the user releases the mouse button.
0x0008	Specifies the focus state. A nonzero value indicates that the button has the focus.

Example

This example determines whether a button currently has the focus:

```
#define BFFOCUS 0x0008

DWORD dwResult;

dwResult = SendDlgItemMessage(hdlg, ID_MYBUTTON, BM_GETSTATE, 0, 0L);
if (dwResult & BFFOCUS)

    /* button has the focus */
```

See Also

BM_GETCHECK, BM_SETSTATE

BM_SETCHECK

2.x

```
BM_SETCHECK
wParam = (WPARAM) fCheck; /* check state            */
lParam = 0L;              /* not used, must be zero */
```

An application sends a BM_SETCHECK message to set the check state of a button.

Parameters

fCheck

Value of *wParam*. Specifies the check state. This parameter can be one of the following values:

Value	Meaning
0	Set the button state to unchecked.
1	Set the button state to checked.
2	Set the button state to indeterminate. This value can be used only if the button has the BS_3STATE or BS_AUTO3STATE style.

Return Value

The return value is always zero.

Comments

The BM_SETCHECK message has no effect on push buttons.

Example

This example places a dot inside a radio button:

```
SendDlgItemMessage(hdlg, ID_MYRADIOBUTTON, BM_SETCHECK, TRUE, 0L);
```

See Also

BM_GETCHECK, BM_GETSTATE, BM_SETSTATE

BM_SETSTATE

2.x

```
BM_SETSTATE
wParam = (WPARAM) fState;    /* highlight state        */
lParam = 0L;                 /* not used, must be zero */
```

An application sends a BM_SETSTATE message to set the highlight state of a button.

Parameters

fState

Value of *wParam*. Specifies whether the button is to be highlighted. A nonzero value highlights the button. A zero value removes any highlighting.

Return Value

The return value is always zero.

Comments

Highlighting affects the exterior of a button. It has no effect on the check state of a radio button or check box.

A button is automatically highlighted when the user presses and holds the left mouse button. The highlighting is removed when the user releases the mouse button.

Example

This example highlights and then removes highlighting from a push button, simulating the visual effect of a user clicking the button:

```
SendDlgItemMessage(hdlg, ID_MYPUSHBUTTON, BM_SETSTATE, TRUE, 0L);

    /*
     * Perform some action; then remove the highlighting,
     * thereby returning it to its normal state.
     */

SendDlgItemMessage(hdlg, ID_MYPUSHBUTTON, BM_SETSTATE, FALSE, 0L);
```

See Also

BM_GETSTATE, BM_SETCHECK

BM_SETSTYLE

2.x

```
BM_SETSTYLE
wParam =  (WPARAM) LOWORD(dwStyle); /* style       */
lParam =  MAKELPARAM(fRedraw, 0);   /* redraw flag */
```

An application sends a BM_SETSTYLE message to change the style of a button.

Parameters

dwStyle
Value of *wParam*. Specifies the button style. For an explanation of button styles, see the following Comments section.

fRedraw
Value of the low-order word of *lParam*. Specifies whether the button is to be redrawn. A value of TRUE redraws the button. A value of FALSE does not redraw the button.

Return Value

The return value is always zero.

Comments

The following are the button styles:

Value	Meaning
BS_3STATE	Creates a button that is the same as a check box, except that the box can be grayed (dimmed) as well as checked. The grayed state typically is used to show that a check box has been disabled.
BS_AUTO3STATE	Creates a button that is the same as a three-state check box, except that the box changes its state when the user selects it. The state cycles through checked, grayed, and normal.
BS_AUTOCHECKBOX	Creates a button that is the same as a check box, except that an X appears in the check box when the user selects the box; the X disappears (is cleared) the next time the user selects the box.
BS_AUTORADIOBUTTON	Creates a button that is the same as a radio button, except that when the user selects it, the button automatically highlights itself and clears (removes the selection from) any other buttons in the same group.
BS_CHECKBOX	Creates a small square that has text displayed to its right (unless this style is combined with the BS_LEFTTEXT style).
BS_DEFPUSHBUTTON	Creates a button that has a heavy black border. The user can select this button by pressing the ENTER key. This style is useful for enabling the user to quickly select the most likely option (the default option).

Value	Meaning
BS_GROUPBOX	Creates a rectangle in which other buttons can be grouped. Any text associated with this style is displayed in the rectangle's upper-left corner.
BS_LEFTTEXT	Places text on the left side of the radio button or check box when combined with a radio button or check box style.
BS_OWNERDRAW	Creates an owner-drawn button. The owner window receives a WM_MEASUREITEM message when the button is created, and it receives a WM_DRAWITEM message when a visual aspect of the button has changed. The BS_OWNERDRAW style cannot be combined with any other button styles.
BS_PUSHBUTTON	Creates a push button that posts a WM_COMMAND message to the owner window when the user selects the button.
BS_RADIOBUTTON	Creates a small circle that has text displayed to its right (unless this style is combined with the BS_LEFTTEXT style). Radio buttons are usually used in groups of related but mutually exclusive choices.

An application should not attempt to change a button's type (for example, changing a radio button to a check box).

Example

This example sends a BM_SETSTYLE message to make a button become the default push button:

```
SendDlgItemMessage(hdlg, ID_MYPUSHBUTTON, BM_SETSTYLE,
    (WPARAM) BS_DEFPUSHBUTTON, TRUE);
```

CB_ADDSTRING

3.0

```
CB_ADDSTRING
wParam = 0;                          /* not used, must be zero   */
lParam = (LPARAM) (LPCSTR) lpsz;     /* address of string to add */
```

An application sends a CB_ADDSTRING message to add a string to the list box of a combo box. If the list box does not have the CBS_SORT style, the string is added to the end of the list. Otherwise, the string is inserted into the list and the list is sorted.

Parameters

lpsz
Value of *lParam*. Points to the null-terminated string to be added. If the combo box was created with an owner-drawn style but without the CBS_HASSTRINGS style, the value of the *lpsz* parameter is stored rather than the string it would otherwise point to.

Return Value

The return value is the zero-based index to the string in the list box. The return value is CB_ERR if an error occurs; the return value is CB_ERRSPACE if insufficient space is available to store the new string.

Comments

If an owner-drawn combo box was created with the CBS_SORT style but not the CBS_HASSTRINGS style, the WM_COMPAREITEM message is sent one or more times to the owner of the combo box so that the new item can be properly placed in the list box.

To insert a string into a specific location within the list, use the CB_INSERTSTRING message.

Example

This example adds the string "my string" to a list box:

```
DWORD dwIndex;

dwIndex = SendDlgItemMessage(hdlg, ID_MYCOMBOBOX,
    CB_ADDSTRING, 0, (LPARAM) ((LPCSTR) "my string"));
```

See Also

CB_INSERTSTRING, WM_COMPAREITEM

CB_DELETESTRING

3.0

```
CB_DELETESTRING
wParam = (WPARAM) index;  /* item to delete         */
lParam = 0L;              /* not used, must be zero */
```

An application sends a CB_DELETESTRING message to delete a string in the list box of a combo box.

Parameters

index
Value of *wParam*. Specifies the zero-based index of the string to delete.

Return Value

The return value is a count of the strings remaining in the list. The return value is CB_ERR if the *index* parameter specifies an index greater than the number of items in the list.

Comments

If the combo box was created with an owner-drawn style but without the CBS_HASSTRINGS style, a WM_DELETEITEM message is sent to the owner of the combo box so that the application can free any additional data associated with the item.

Example

This example deletes the first string in a combo box:

```
DWORD dwRemaining;

dwRemaining = SendDlgItemMessage(hdlg, ID_MYCOMBOBOX,
    CB_DELETESTRING, 0, 0L);
```

See Also

WM_DELETEITEM

CB_DIR

3.0

```
CB_DIR
wParam = (WPARAM) (UINT) uAttrs;         /* file attributes     */
lParam = (LPARAM) (LPCSTR) lpszFileSpec; /* address of filename */
```

An application sends a CB_DIR message to add a list of filenames to the list box of a combo box.

Parameters

uAttrs

Value of *wParam*. Specifies the attributes of the files to be added to the list box. It can be any combination of the following values:

Value	Meaning
0x0000	File can be read from or written to.
0x0001	File can be read from but not written to.
0x0002	File is hidden and does not appear in a directory listing.
0x0004	File is a system file.
0x0010	The name pointed to by the *lpszFileSpec* parameter specifies a directory.
0x0020	File has been archived.
0x4000	All drives that match the name specified by the *lpszFileSpec* parameter are included.
0x8000	Exclusive flag. If the exclusive flag is set, only files of the specified type are listed. Otherwise, files of the specified type are listed in addition to files that do not match the specified type.

lpszFileSpec
Value of *lParam*. Points to the null-terminated string that specifies the filename to add to the list. If the filename contains any wildcards (for example, *.*), all files that match and have the attributes specified by the *uAttrs* parameter will be added to the list.

Return Value

The return value is the zero-based index of the last filename added to the list. The return value is CB_ERR if an error occurs. The return value is CB_ERRSPACE if insufficient space is available to store the new strings.

Example

This example adds the names of all available drives to a combo box:

```
DWORD dwIndexLastItem;

dwIndexLastItem = SendDlgItemMessage(hdlg, ID_MYCOMBOBOX, CB_DIR,
    0x4000 | 0x8000, (LPARAM) ((LPCSTR) "*"));
```

See Also

DlgDirList

CB_FINDSTRING

3.0

```
CB_FINDSTRING
wParam = (WPARAM) indexStart;        /* item before start of search */
lParam = (LPARAM) (LPCSTR) lpszFind; /* address of prefix string    */
```

An application sends a CB_FINDSTRING message to find the first string that contains the prefix specified in the list box of a combo box.

Parameters

indexStart
Value of *wParam*. Specifies the zero-based index of the item before the first item to be searched. When the search reaches the bottom of the list box, it continues from the top of the list box back to the item specified by the *indexStart* parameter. If *indexStart* is –1, the entire list box is searched from the beginning.

lpszFind
Value of *lParam*. Points to the null-terminated string that contains the prefix to search for. The search is not case-sensitive, so this string can contain any combination of uppercase and lowercase letters.

Return Value

The return value is the zero-based index of the matching item, or it is CB_ERR if the search was unsuccessful.

Comments

If the combo box's style is owner-drawn but not CBS_HASSTRINGS and CBS_SORT, CB_FINDSTRING is used. If the styles are owner-drawn and CBS_SORT but not CBS_HASSTRINGS, WM_COMPAREITEM messages are sent.

Example

This example searches for the string "my string" in a combo box and copies it, if found, to the szBuf buffer:

```
char szBuf[20];
DWORD dwIndex;

dwIndex = SendDlgItemMessage(hdlg, ID_MYCOMBOBOX,
    CB_FINDSTRING, 0, (LPARAM) ((LPCSTR) "my string"));
if (dwIndex != CB_ERR)
    SendDlgItemMessage(hdlg, ID_MYCOMBOBOX,
        CB_GETLBTEXT, (WPARAM) dwIndex, (LPARAM) ((LPCSTR) szBuf));
```

See Also

CB_FINDSTRINGEXACT, CB_SETCURSEL

CB_FINDSTRINGEXACT

3.1

```
CB_FINDSTRINGEXACT
wParam = (WPARAM) indexStart;        /* item before start of search */
lParam = (LPARAM) (LPCSTR) lpszFind; /* address of prefix string    */
```

An application sends a CB_FINDSTRINGEXACT message to find the first list box string (in a combo box) that matches the string specified in the *lpszFind* parameter.

Parameters

indexStart

Value of *wParam*. Specifies the zero-based index of the item before the first item to be searched. When the search reaches the bottom of the list box, it continues from the top of the list box back to the item specified by the *indexStart* parameter. If *indexStart* is −1, the entire list box is searched from the beginning.

lpszFind

Value of *lParam*. Points to the null-terminated string to search for. This string can contain a complete filename, including the extension. The search is not case-sensitive, so this string can contain any combination of uppercase and lowercase letters.

Return Value

The return value is the zero-based index of the matching item, or it is CB_ERR if the search was unsuccessful.

Comments If the combo box's style is owner-drawn but not CBS_HASSTRINGS and CBS_SORT, CB_FINDSTRING is used. If the styles are owner-drawn and CBS_SORT but not CBS_HASSTRINGS, WM_COMPAREITEM messages are sent.

See Also CB_FINDSTRING, CB_SETCURSEL

CB_GETCOUNT

3.0

```
CB_GETCOUNT
wParam = 0;      /* not used, must be zero */
lParam = 0L;     /* not used, must be zero */
```

An application sends a CB_GETCOUNT message to retrieve the number of items in the list box of a combo box.

Parameters This message has no parameters.

Return Value The return value is the number of items in the list box.

Comments The returned count is one greater than the index value of the last item (the index is zero-based).

Example This example retrieves the number of items in a combo box:

```
WORD cListItems;

cListItems = (WORD) SendDlgItemMessage(hdlg, ID_MYCOMBOBOX,
    CB_GETCOUNT, 0, 0);
```

CB_GETCURSEL

3.0

```
CB_GETCURSEL
wParam = 0;      /* not used, must be zero */
lParam = 0L;     /* not used, must be zero */
```

An application sends a CB_GETCURSEL message to retrieve the index of the currently selected item, if any, in the list box of a combo box.

Parameters This message has no parameters.

Return Value The return value is the zero-based index of the currently selected item, or it is CB_ERR if no item is selected.

Example This example retrieves the index of the currently selected string in the list box of a combo box and then retrieves that string:

```
char szBuf[20];
DWORD dwIndex;

dwIndex = SendDlgItemMessage(hdlg, ID_MYCOMBOBOX, CB_GETCURSEL, 0, 0);
if (dwIndex != CB_ERR)
    SendDlgItemMessage(hdlg, ID_MYCOMBOBOX,
        CB_GETLBTEXT, (WPARAM) dwIndex, (LPARAM) ((LPCSTR) szBuf));
```

See Also CB_SETCURSEL

CB_GETDROPPEDCONTROLRECT

3.1

```
CB_GETDROPPEDCONTROLRECT
wParam = 0;                              /* not used, must be zero    */
lParam = (LPARAM) (RECT FAR*) lprc;      /* address of RECT structure */
```

An application sends a CB_GETDROPPEDCONTROLRECT message to retrieve the screen coordinates of the visible (dropped-down) list box of a combo box.

Parameters *lprc*

Value of *lParam*. Points to the **RECT** structure that is to receive the coordinates. The **RECT** structure has the following form:

```
typedef struct tagRECT {    /* rc */
   int left;
   int top;
   int right;
   int bottom;
} RECT;
```

Return Value The return value is always CB_OKAY.

Example This example retrieves the bounding rectangle of the list box of a combo box:

```
RECT rcl;

SendDlgItemMessage(hdlg, ID_MYCOMBOBOX,
    CB_GETDROPPEDCONTROLRECT, 0, (DWORD) ((LPRECT) &rcl));
```

CB_GETDROPPEDSTATE

3.1

```
CB_GETDROPPEDSTATE
wParam = 0;      /* not used, must be zero */
lParam = 0L;     /* not used, must be zero */
```

An application sends a CB_GETDROPPEDSTATE message to determine whether the list box of a combo box is visible (dropped down).

Parameters This message has no parameters.

Return Value The return value is nonzero if the list box is visible; otherwise, it is zero.

Example This example determines whether the list box of a combo box is visible:

```
BOOL fDropped;

fDropped = (BOOL) SendDlgItemMessage(hdlg, ID_MYCOMBOBOX,
    CB_GETDROPPEDSTATE, 0, 0L);
```

See Also CB_SHOWDROPDOWN

CB_GETEDITSEL

2.x

```
CB_GETEDITSEL
wParam = 0;      /* not used, must be zero */
lParam = 0L;     /* not used, must be zero */
```

An application sends a CB_GETEDITSEL message to retrieve the starting and ending character positions of the current selection in the edit control of a combo box.

Parameters This message has no parameters.

Return Value The return value is a doubleword value that contains the starting position in the low-order word and the position of the first nonselected character after the end of the selection in the high-order word.

Example This example retrieves the selection positions of the edit control of a combo box, and converts them into starting and ending positions:

```
DWORD dwResult;
WORD wStart, wEnd;

dwResult = SendDlgItemMessage(hdlg, ID_MYCOMBOBOX,
        CB_GETEDITSEL, 0, 0L);
wStart = LOWORD(dwResult);
wEnd   = HIWORD(dwResult);
```

See Also CB_SETEDITSEL

CB_GETEXTENDEDUI

3.1

```
CB_GETEXTENDEDUI
wParam = 0;      /* not used, must be zero */
lParam = 0L;     /* not used, must be zero */
```

An application sends a CB_GETEXTENDEDUI message to determine whether a combo box has the default user interface or the extended user interface.

Parameters This message has no parameters.

Return Value The return value is nonzero if the combo box has the extended user interface; otherwise, it is zero.

Comments The extended user interface differs from the default user interface in the following ways:

- Clicking the static control displays the list box (CBS_DROPDOWNLIST style only).
- Pressing the DOWN ARROW key displays the list box (F4 is disabled).
- Scrolling in the static control is disabled when the item list is not visible (arrow keys are disabled).

Example

This example determines whether a combo box has the extended user interface:

```
BOOL fExtended;

fExtended = (BOOL) SendDlgItemMessage(hdlg, ID_MYCOMBOBOX,
    CB_GETEXTENDEDUI, 0, 0L);
```

See Also

CB_SETEXTENDEDUI

CB_GETITEMDATA

3.0

```
CB_GETITEMDATA
wParam = (WPARAM) index;  /* item index             */
lParam = 0L;              /* not used, must be zero */
```

An application sends a CB_GETITEMDATA message to a combo box to retrieve the application-supplied doubleword value associated with the specified item in the combo box. (This is the value in the *lParam* parameter of a CB_SETITEMDATA message.)

Parameters

index
Value of *wParam*. Specifies the zero-based index of the item.

Return Value

The return value is the doubleword value associated with the item, or it is CB_ERR if an error occurs.

See Also

CB_SETITEMDATA

CB_GETITEMHEIGHT

3.1

```
CB_GETITEMHEIGHT
wParam = (WPARAM) index;  /* item index             */
lParam = 0L;              /* not used, must be zero */
```

An application sends a CB_GETITEMHEIGHT message to retrieve the height of list items in a combo box.

Parameters

index
Value of *wParam*. Specifies the component of the combo box whose height is to be retrieved. If the *index* parameter is –1, the height of the edit-control (or static-text) portion of the combo box is retrieved. If the combo box has the CBS_OWNERDRAWVARIABLE style, *index* specifies the zero-based index of the list item whose height is to be retrieved. Otherwise, *index* should be set to zero.

Return Value

The return value is the height, in pixels, of the list items in a combo box. The return value is the height of the item specified by the *index* parameter if the combo box has the CBS_OWNERDRAWVARIABLE style. The return value is the height of the edit-control (or static-text) portion of the combo box if *index* is –1. The return value is CB_ERR if an error occurred.

Example

This example sends a CB_GETITEMHEIGHT message to retrieve the height of the list items in a combo box:

```
LRESULT lrHeight;

lrHeight = SendDlgItemMessage(hdlg, ID_MYCOMBOBOX,
    CB_GETITEMHEIGHT, 0, 0L);
```

See Also

CB_SETITEMHEIGHT

CB_GETLBTEXT

3.0

```
CB_GETLBTEXT
wParam = (WPARAM) index;                /* item index        */
lParam = (LPARAM) (LPCSTR) lpszBuffer;  /* address of buffer */
```

An application sends a CB_GETLBTEXT message to retrieve a string from the list box of a combo box.

Parameters

index
Value of *wParam*. Specifies the zero-based index of the string to retrieve.

lpszBuffer
Value of *lParam*. Points to the buffer that receives the string. The buffer must have sufficient space for the string and a terminating null character. A CB_GETLBTEXTLEN message can be sent before the CB_GETLBTEXT message to retrieve the length, in bytes, of the string.

Return Value

The return value is the length of the string, in bytes, excluding the terminating null character. If the *index* parameter does not specify a valid index, the return value is CB_ERR.

Comments

If the combo box was created with an owner-drawn style but without the CBS_HASSTRINGS style, the buffer pointed to by the *lpszBuffer* parameter of the message receives the doubleword value associated with the item.

Example

This example retrieves the length of the first item in the list box of a combo box, allocates sufficient memory for the string, and sends a CB_GETLBTEXT message to retrieve the string:

```
DWORD cbItemString;
PSTR psz;

cbItemString = SendDlgItemMessage(hdlg, ID_MYCOMBOBOX,
    CB_GETLBTEXTLEN, 0, 0L);
if (cbItemString != CB_ERR) {
    psz = (PSTR) LocalAlloc(LMEM_FIXED, (WORD) cbItemString);
    SendDlgItemMessage(hdlg, ID_MYCOMBOBOX,
        CB_GETLBTEXT, 0, (LPARAM) ((LPCSTR) psz));
}
```

See Also

CB_GETLBTEXTLEN

CB_GETLBTEXTLEN

3.0

```
CB_GETLBTEXTLEN
wParam = (WPARAM) index;  /* item index             */
lParam = 0L;              /* not used, must be zero */
```

An application sends a CB_GETLBTEXTLEN message to retrieve the length of a string in the list box of a combo box.

Parameters

index

Value of *wParam*. Specifies the zero-based index of the string.

Return Value

The return value is the length of the string, in bytes, excluding the terminating null character. If the *index* parameter does not specify a valid index, the return value is CB_ERR.

Example

This example retrieves the length of the first item in the list box of a combo box:

```
DWORD cbItemString;

cbItemString = SendDlgItemMessage(hdlg, ID_MYCOMBOBOX,
    CB_GETLBTEXTLEN, 0, 0L);
```

See Also

CB_GETLBTEXT

CB_INSERTSTRING

3.0

```
CB_INSERTSTRING
wParam = (WPARAM) index;            /* item index                  */
lParam = (LPARAM) (LPCSTR) lpsz;    /* address of string to insert */
```

An application sends a CB_INSERTSTRING message to insert a string into the list box of a combo box. Unlike the CB_ADDSTRING message, the CB_INSERTSTRING message does not cause a list with the CBS_SORT style to be sorted.

Parameters

index

Value of *wParam*. Specifies the zero-based index of the position at which to insert the string. If this parameter is –1, the string is added to the end of the list.

lpsz

Value of *lParam*. Points to the null-terminated string that is to be inserted. If the combo box was created with an owner-drawn style but without the CBS_HASSTRINGS style, the value of the *lpsz* parameter is stored rather than the string it would otherwise point to.

Return Value

The return value is the index of the position at which the string was inserted. The return value is CB_ERR if an error occurs. The return value is CB_ERRSPACE if insufficient space is available to store the new string.

Example

This example inserts the string "my string" into the third position in the list box of a combo box:

```
SendDlgItemMessage(hdlg, ID_MYCOMBOBOX,
    CB_INSERTSTRING, 2, (LPARAM) ((LPCSTR) "my string"));
```

See Also

CB_ADDSTRING

CB_LIMITTEXT

3.0

```
CB_LIMITTEXT
wParam = (WPARAM) cchLimit;    /* maximum number of characters */
lParam = 0L;                   /* not used, must be zero       */
```

An application sends a CB_LIMITTEXT message to limit the length of the text that the user may type in the edit control of a combo box.

Parameters

cchLimit
Value of *wParam*. Specifies the length, in bytes, of the text the user can enter. If this parameter is zero, the text length is set to 65,535 bytes.

Return Value

The return value is 1 if the message is successful. If this message is sent to a combo box with the style CBS_DROPDOWNLIST, the return value is CB_ERR.

Comments

If the combo box does not have the style CBS_AUTOHSCROLL, setting the text limit to be larger than the size of the edit control has no effect.

The CB_LIMITTEXT message limits only the text the user can enter. It has no effect on any text already in the edit control when the message is sent, nor does it affect the length of the text copied to the edit control when a string in the list box is selected.

Example

This example limits the text of the edit control of a combo box to five characters:

```
SendDlgItemMessage(hdlg, ID_MYCOMBOBOX, CB_LIMITTEXT, 5, 0L);
```

CB_RESETCONTENT

3.0

```
CB_RESETCONTENT
wParam = 0;      /* not used, must be zero */
lParam = 0L;     /* not used, must be zero */
```

An application sends a CB_RESETCONTENT message to remove all items from the list box and edit control of a combo box.

Parameters

This message has no parameters.

Return Value

The return value is always CB_OKAY.

Comments

If the combo box was created with an owner-drawn style but without the CBS_HASSTRINGS style, the owner of the combo box receives a WM_DELETEITEM message for each item in the combo box.

Example

This example removes all items from the list box and edit control of a combo box:

```
SendDlgItemMessage(hdlg, ID_MYCOMBOBOX, CB_RESETCONTENT, 0, 0L);
```

See Also

WM_DELETEITEM

CB_SELECTSTRING

3.0

```
CB_SELECTSTRING
wParam = (WPARAM) indexStart;          /* item before first selection */
lParam = (LPARAM) (LPCSTR) lpszSelect; /* address of prefix string    */
```

An application sends a CB_SELECTSTRING message to search for a string in the list box of a combo box and, if the string is found, to select the string in the list box and copy it to the edit control.

Parameters

indexStart

Value of *wParam*. Specifies the zero-based index of the item before the first item to be searched. When the search reaches the bottom of the list box, it continues from the top of the list box back to the item specified by the *indexStart* parameter. If *indexStart* is –1, the entire list box is searched from the beginning.

lpszSelect

Value of *lParam*. Points to the null-terminated string that contains the prefix to search for. The search is not case-sensitive, so this string can contain any combination of uppercase and lowercase letters.

Return Value

The return value is the index of the selected item if the string was found. The return value is CB_ERR and the current selection is not changed if the search was unsuccessful.

Comments

A string is selected only if its initial characters (from the starting point) match the characters in the prefix string.

If the combo box's style is owner-drawn but not CBS_HASSTRINGS and CBS_SORT, CB_FINDSTRING is used. If the styles are owner-drawn and CBS_SORT but not CBS_HASSTRINGS, WM_COMPAREITEM messages are sent.

Example

This example searches the entire list box of a combo box for the string "my string" and, if the string is found, selects it:

```
DWORD dwIndexFoundString;

dwIndexFoundString = SendDlgItemMessage(hdlg, ID_MYCOMBOBOX,
    CB_SELECTSTRING, -1, (LPARAM) ((LPCSTR) "my string"));
```

See Also

CB_FINDSTRING

CB_SETCURSEL

3.0

```
CB_SETCURSEL
wParam = (WPARAM) index;  /* item index              */
lParam = 0L;              /* not used, must be zero */
```

An application sends a CB_SETCURSEL message to select a string in the list box of a combo box. If necessary, the list box scrolls the string into view (if the list box is visible). The text in the edit control of the combo box is changed to reflect the new selection. Any previous selection in the list box is removed.

Parameters

index

Value of *wParam*. Specifies the zero-based index of the string to select. If the *index* parameter is –1, any current selection in the list box is removed and the edit control is cleared.

Return Value

The return value is the index of the item selected if the message is successful. The return value is CB_ERR if the *index* parameter is greater than the number of items in the list or if *index* is set to –1 (which clears the selection).

Example

This example retrieves the number of items in the list box of a combo box and sends a CB_SETCURSEL message to select the last item in the list:

```
WORD cListItems;

cListItems = (WPARAM) SendDlgItemMessage(hdlg,
    ID_MYCOMBOBOX, CB_GETCOUNT, 0, 0);
SendDlgItemMessage(hdlg, ID_MYCOMBOBOX,
    CB_SETCURSEL,
    cListItems - 1,   /* zero-based index, so subtract one from total */
    0L);
```

See Also

CB_GETCURSEL, CB_FINDSTRING

CB_SETEDITSEL

3.0

```
CB_SETEDITSEL
wParam = 0;                                  /* not used, must be zero  */
lParam = MAKELPARAM(ichStart, ichEnd);       /* start and end positions */
```

An application sends a CB_SETEDITSEL message to select characters in the edit control of a combo box.

Parameters

ichStart
Value of the low-order word of *lParam*. Specifies the starting position. If this parameter is set to −1, the selection, if any, is removed.

ichEnd
Value of the high-order word of *lParam*. Specifies the ending position. If this parameter is set to −1, all text from the starting position to the last character in the edit control is selected.

Return Value

The return value is nonzero if the message is successful. It is CB_ERR if the message is sent to a combo box with the CBS_DROPDOWNLIST style.

Comments

The positions are zero-based. To select the first character of the edit control, you specify a starting position of zero. The ending position is for the character just after the last character to select. For example, to select the first four characters of the edit control, you would use a starting position of 0 and an ending position of 4.

Example

This example selects the first four characters of the edit control of a combo box:

```
SendDlgItemMessage(hdlg, ID_MYCOMBOBOX,
    CB_SETEDITSEL, 0, MAKELONG(0, 4));
```

See Also

CB_GETEDITSEL

CB_SETEXTENDEDUI

3.1

```
CB_SETEXTENDEDUI
wParam = (WPARAM) (BOOL) fExtended;      /* extended UI flag       */
lParam = 0L;                             /* not used, must be zero */
```

An application sends a CB_SETEXTENDEDUI message to select either the default user interface or the extended user interface for a combo box that has the CBS_DROPDOWN or CBS_DROPDOWNLIST style.

Parameters

fExtended
Value of *wParam*. Specifies whether the combo box should use the extended user interface or the default user interface. A value of TRUE selects the extended user interface; a value of FALSE selects the standard user interface.

Return Value

The return value is CB_OKAY if the operation is successful, or it is CB_ERR if an error occurred.

Comments

The extended user interface differs from the default user interface in the following ways:

- Clicking the static control displays the list box (CBS_DROPDOWNLIST style only).
- Pressing the DOWN ARROW key displays the list box (F4 is disabled).
- Scrolling in the static control is disabled when the item list is not visible (the arrow keys are disabled).

Example

This example selects the extended user interface for a combo box:

```
SendDlgItemMessage(hdlg, ID_MYCOMBOBOX, CB_SETEXTENDEDUI,
    TRUE, 0L);
```

See Also

CB_GETEXTENDEDUI

CB_SETITEMDATA

3.0

```
CB_SETITEMDATA
wParam = (WPARAM) index;            /* item index */
lParam = (LPARAM) (DWORD) dwData;   /* item data  */
```

An application sends a CB_SETITEMDATA message to set the doubleword value associated with the specified item in a combo box. If the item is in an owner-drawn combo box created without the CBS_HASSTRINGS style, this message replaces the doubleword value that was contained in the *lParam* parameter of the CB_ADDSTRING or CB_INSERTSTRING message that added the item to the combo box.

Parameters

index
Value of *wParam*. Specifies the zero-based index to the item.

dwData
Value of *lParam*. Specifies the new value to be associated with the item.

Return Value The return value is CB_ERR if an error occurs.

See Also CB_ADDSTRING, CB_INSERTSTRING

CB_SETITEMHEIGHT

3.1

```
CB_SETITEMHEIGHT
wParam = (WPARAM) index;        /* item index  */
lParam = (LPARAM) (int) height; /* item height */
```

An application sends a CB_SETITEMHEIGHT message to set the height of list items in a combo box or the height of the edit-control (or static-text) portion of a combo box.

Parameters

index
Value of *wParam*. Specifies whether the height of list items or the height of the edit-control (or static-text) portion of the combo box is set.

If the combo box has the CBS_OWNERDRAWVARIABLE style, the *index* parameter specifies the zero-based index of the list item whose height is to be set; otherwise, *index* must be zero and the height of all list items will be set.

If *index* is –1, the height of the edit-control or static-text portion of the combo box is to be set.

height
Value of the low-order word of *lParam*. Specifies the height, in pixels, of the combo box component identified by *index*.

Return Value The return value is CB_ERR if the index or height is invalid.

Comments The height of the edit-control (or static-text) portion of the combo box is set independently of the height of the list items. An application must ensure that the height of the edit-control (or static-text) portion isn't smaller than the height of a particular list box item.

Example This example sends a CB_SETITEMHEIGHT message to set the height of list items in a combo box:

```
LPARAM lrHeight;

SendDlgItemMessage(hdlg, ID_MYCOMBOBOX, CB_SETITEMHEIGHT,
    0, lrHeight);
```

See Also CB_GETITEMHEIGHT

CB_SHOWDROPDOWN

3.0

```
CB_SHOWDROPDOWN
wParam = (WPARAM) (BOOL) fShow;      /* the show/hide flag     */
lParam = 0L;                         /* not used, must be zero */
```

An application sends a CB_SHOWDROPDOWN message to show or hide the list box of a combo box that has the CBS_DROPDOWN or CBS_DROPDOWNLIST style.

Parameters

fShow
Value of *wParam*. Specifies whether the drop-down list box is to be shown or hidden. A value of TRUE shows the list box. A value of FALSE hides the list box.

Return Value

The return value is always nonzero.

Comments

This message has no effect on a combo box created with the CBS_SIMPLE style.

Example

This example shows the list box of a combo box:

```
SendDlgItemMessage(hdlg, ID_MYCOMBOBOX, CB_SHOWDROPDOWN, TRUE, 0L);
```

DM_GETDEFID

```
DM_GETDEFID
wParam = 0;      /* not used, must be zero */
lParam = 0L;     /* not used, must be zero */
```

An application sends a DM_GETDEFID message to get the identifier of the default push button for a dialog box.

Parameters

This message has no parameters.

Return Value

The return value is a doubleword value. If the default push button has an identifier value, the high-order word contains DC_HASDEFID and the low-order word contains the identifier value. The return value is zero if the default push button does not have an identifier value.

Example

This example gets the identifier of the default push button of a dialog box:

```
DWORD dwResult;
WORD idDefPushButton;

dwResult = SendMessage(hdlg, DM_GETDEFID, 0, 0L);
if (HIWORD(dwResult) == DC_HASDEFID)
    idDefPushButton = LOWORD(dwResult);
```

See Also

DM_SETDEFID

DM_SETDEFID

2.x

```
DM_SETDEFID
wIDPushBtn = wParam; /* identifier of new default push button */
```

An application sends a DM_SETDEFID message to change the identifier of the default push button for a dialog box.

Parameters

wIDPushBtn
Value of *wParam*. Specifies the identifier of the push button that will become the default.

Return Value

The return value is always nonzero.

EM_CANUNDO

2.x

```
EM_CANUNDO
wParam = 0;     /* not used, must be zero */
lParam = 0L;    /* not used, must be zero */
```

An application sends an EM_CANUNDO message to determine whether an edit-control operation can be undone.

Parameters

This message has no parameters.

Return Value

The return value is nonzero if the last edit operation can be undone, or it is zero if the last edit operation cannot be undone.

Example

This example sends an EM_CANUNDO message to determine whether the last edit-control operation can be undone and, if so, sends an EM_UNDO message to undo the last operation:

```
if (SendDlgItemMessage(hdlg, ID_MYEDITCONTROL, EM_CANUNDO, 0, 0L))
    SendDlgItemMessage(hdlg, ID_MYEDITCONTROL, EM_UNDO, 0, 0L);
```

See Also

EM_UNDO

EM_EMPTYUNDOBUFFER

3.0

```
EM_EMPTYUNDOBUFFER
wParam = 0;     /* not used, must be zero */
lParam = 0L;    /* not used, must be zero */
```

An application sends an EM_EMPTYUNDOBUFFER message to reset (clear) the undo flag of an edit control. The undo flag is set whenever an operation within the edit control can be undone.

Parameters

This message has no parameters.

Return Value

This message does not return a value.

Comments

The undo flag is automatically cleared whenever the edit control receives a WM_SETTEXT or EM_SETHANDLE message.

Example

This example resets the undo flag of an edit control:

```
SendDlgItemMessage(hdlg, ID_MYEDITCONTROL, EM_EMPTYUNDOBUFFER, 0, 0L);
```

See Also

EM_CANUNDO, EM_SETHANDLE, EM_UNDO, WM_SETTEXT

EM_FMTLINES

2.x

```
EM_FMTLINES
wParam = (WPARAM) (BOOL) fAddEOL;   /* line break flag        */
lParam = 0L;                        /* not used, must be zero */
```

An application sends an EM_FMTLINES message to set the inclusion of soft line break characters on or off within a multiline edit control. A soft line break consists of two carriage returns and a linefeed inserted at the end of a line that is broken because of wordwrapping.

This message is processed only by multiline edit controls.

Parameters

fAddEOL
Value of *wParam*. Specifies whether soft line break characters are to be inserted. A value of TRUE inserts the characters; a value of FALSE removes them.

Return Value

The return value is identical to the *fAddEOL* parameter.

Comments

This message affects only the buffer returned by the EM_GETHANDLE message and the text returned by the WM_GETTEXT message. It has no effect on the display of the text within the edit control.

A line that ends with a hard line break is not affected by the EM_FMTLINES message. A hard line break consists of one carriage return and a linefeed.

Example

This example sends an EM_FMTLINES message to turn off soft line breaks, then allocates a buffer for the text, and then retrieves the text by sending a WM_GETTEXT message:

```
WPARAM cbText;
HGLOBAL hmem;
LPSTR lpstr;

SendDlgItemMessage(hdlg, ID_MYEDITCONTROL,
    EM_FMTLINES, FALSE, 0);

cbText = (WPARAM) SendDlgItemMessage(hdlg, ID_MYEDITCONTROL,
    WM_GETTEXTLENGTH, 0, 0L);
cbText++;      /* make room for the terminating null character */
hmem = (HGLOBAL) GlobalAlloc(GMEM_MOVEABLE, (DWORD) cbText);
lpstr = GlobalLock(hmem);
SendDlgItemMessage(hdlg, ID_MYEDITCONTROL,
    WM_GETTEXT, cbText, (LPARAM) lpstr);
```

See Also

EM_GETHANDLE, WM_GETTEXT

EM_GETFIRSTVISIBLELINE

3.1

```
EM_GETFIRSTVISIBLELINE
wParam = 0;      /* not used, must be zero */
lParam = 0L;     /* not used, must be zero */
```

An application sends an EM_GETFIRSTVISIBLELINE message to determine the topmost visible line in an edit control.

Parameters

This message has no parameters.

Return Value

The return value is the zero-based index of the topmost visible line. For single-line edit controls, the return value is zero.

Example

This example gets the index of the topmost visible line in an edit control:

```
int FirstVis;

FirstVis = (int) SendDlgItemMessage(hdlg, IDD_EDIT,
    EM_GETFIRSTVISIBLELINE, 0, 0L);
```

EM_GETHANDLE

2.x

```
EM_GETHANDLE
wParam = 0;      /* not used, must be zero */
lParam = 0L;     /* not used, must be zero */
```

An application sends an EM_GETHANDLE message to retrieve a handle to the memory currently allocated for a multiline edit control. The handle is a local memory handle and can be used by any of the functions that take a local memory handle as a parameter.

This message is processed only by multiline edit controls.

Parameters

This message has no parameters.

Return Value

The return value is a local memory handle identifying the buffer that holds the contents of the edit control. If an error occurs, such as sending the message to a single-line edit control, the return value is zero.

Comments

An application can send this message to a multiline edit control in a dialog box only if it created the dialog box with the DS_LOCALEDIT style flag set. If the

DS_LOCALEDIT style is not set, the return value is still nonzero, but the return value will not be meaningful.

Example

This example sends an EM_GETHANDLE message to a multiline edit control and calls the **LocalSize** function to determine the current size of the edit control using the handle returned by the EM_GETHANDLE message:

```
HANDLE hmemMle;
WORD cbMle;

hmemMle = (HLOCAL) SendDlgItemMessage(hdlg, ID_MYEDITCONTROL,
    EM_GETHANDLE, 0, 0L);
cbMle = LocalSize(hmemMle);
```

See Also

EM_SETHANDLE

EM_GETLINE

2.x

```
EM_GETLINE
wParam = (WPARAM) line;         /* line number to retrieve    */
lParam = (LPARAM) (LPSTR) lpch; /* address of buffer for line */
```

An application sends an EM_GETLINE message to retrieve a line of text from an edit control.

Parameters

line

Value of *wParam*. Specifies the line number of the line to retrieve from a multiline edit control. Line numbers are zero-based; a value of zero specifies the first line. This parameter is ignored by a single-line edit control.

lpch

Value of *lParam*. Points to the buffer that receives a copy of the line. The first word of the buffer specifies the maximum number of bytes that can be copied to the buffer.

Return Value

The return value is the number of bytes actually copied. The return value is zero if the line number specified by the *line* parameter is greater than the number of lines in the edit control.

Comments

The copied line does not contain a terminating null character.

Example

This example sets the maximum size of the buffer, sends an EM_GETLINE message to get the first line of the multiline edit control, and adds a terminating null character to the end of the retrieved line:

```
unsigned char szBuf[128];
WORD cbText;

*(WORD *) szBuf = sizeof(szBuf) - 1; /* sets the buffer size   */
cbText = (WORD) SendDlgItemMessage(hdlg, ID_MYEDITCONTROL,
    EM_GETLINE,
    0,                              /* line number             */
    (DWORD) (LPSTR) szBuf);         /* buffer address          */
szBuf[cbText] = '\0';               /* terminating null character */
```

See Also

EM_LINELENGTH, WM_GETTEXT

EM_GETLINECOUNT

2.x

```
EM_GETLINECOUNT
wParam = 0;     /* not used, must be zero */
lParam = 0L;    /* not used, must be zero */
```

An application sends an EM_GETLINECOUNT message to retrieve the number of lines in a multiline edit control.

This message is processed only by multiline edit controls.

Parameters

This message has no parameters.

Return Value

The return value is an integer containing the number of lines in the multiline edit control. If no text is in the edit control, the return value is 1.

Example

This example sends an EM_GETLINECOUNT message to retrieve the number of lines in a multiline edit control and then sends an EM_LINESCROLL message to scroll the edit control so that the last line is displayed at the top of the edit control.

```
int cLines;

cLines = (int) SendDlgItemMessage(hdlg, ID_MYEDITCONTROL,
    EM_GETLINECOUNT, 0, 0L);
SendDlgItemMessage(hdlg, ID_MYEDITCONTROL,
    EM_LINESCROLL, 0, MAKELONG(cLines - 1, 0));
```

See Also

EM_GETLINE, EM_LINELENGTH

EM_GETMODIFY

2.x

```
EM_GETMODIFY
wParam = 0;      /* not used, must be zero */
lParam = 0L;     /* not used, must be zero */
```

An application sends an EM_GETMODIFY message to determine whether the contents of an edit control have been modified.

Parameters

This message has no parameters.

Return Value

The return value is nonzero if the edit-control contents have been modified, or it is zero if the contents have remained unchanged.

Comments

Windows maintains an internal flag indicating whether the contents of the edit control have been changed. This flag is cleared when the edit control is first created; or an EM_SETMODIFY message can be sent to clear the flag.

Example

This example sends an EM_GETMODIFY message to determine whether the edit control has been modified and, if it has, retrieves the current contents of the edit control and clears the modification flag by sending an EM_SETMODIFY message:

```
char szBuf[128];

if (SendDlgItemMessage(hdlg, ID_MYEDITCONTROL,
        EM_GETMODIFY, 0, 0L)) {
    SendDlgItemMessage(hdlg, ID_MYEDITCONTROL,
        WM_GETTEXT, sizeof(szBuf), (LPARAM) ((LPCSTR) szBuf));
    SendDlgItemMessage(hdlg, ID_MYEDITCONTROL,
        EM_SETMODIFY, FALSE, 0L);
}
```

See Also

EM_SETMODIFY

EM_GETPASSWORDCHAR

3.1

```
EM_GETPASSWORDCHAR
wParam = 0;      /* not used, must be zero */
lParam = 0L;     /* not used, must be zero */
```

An application sends an EM_GETPASSWORDCHAR message to retrieve the password character displayed in an edit control when the user enters text.

Parameters

This message has no parameters.

Return Value

The return value specifies the character to be displayed in place of the character typed by the user. The return value is NULL if no password character exists.

Comments

If the edit control is created with the ES_PASSWORD style, the default password character is set to an asterisk (*).

See Also

EM_SETPASSWORDCHAR

EM_GETRECT

2.x

```
EM_GETRECT
wParam = 0;                          /* not used, must be zero    */
lParam = (LPARAM) (RECT FAR*) lprc; /* address of RECT structure */
```

An application sends an EM_GETRECT message to retrieve the formatting rectangle of an edit control. The formatting rectangle is the limiting rectangle of the text. The limiting rectangle is independent of the size of the edit-control window.

Parameters

lprc

Value of *lParam*. Points to the **RECT** structure that receives the formatting rectangle. The **RECT** structure has the following form:

```
typedef struct tagRECT {    /* rc */
   int left;
   int top;
   int right;
   int bottom;
} RECT;
```

Return Value

The return value is not a meaningful value.

Comments The formatting rectangle of a multiline edit control can be modified by the EM_SETRECT and EM_SETRECTNP messages.

Example This example sends an EM_GETRECT message to retrieve the formatting rectangle of an edit control:

```
RECT rcl;

SendDlgItemMessage(hdlg, ID_MYEDITCONTROL,
    EM_GETRECT, 0, (DWORD) ((LPRECT) &rcl));
```

See Also EM_SETRECT

EM_GETSEL

2.x

```
EM_GETSEL
wParam = 0;      /* not used, must be zero */
lParam = 0L;     /* not used, must be zero */
```

An application sends an EM_GETSEL message to get the starting and ending character positions of the current selection in an edit control.

Parameters This message has no parameters.

Return Value The return value is a doubleword value that contains the starting position in the low-order word and the position of the first nonselected character after the end of the selection in the high-order word.

Example This example gets the selection positions of an edit control and converts them into starting and ending positions:

```
DWORD dwResult;
WORD wStart, wEnd;

dwResult = SendDlgItemMessage(hdlg, ID_MYCOMBOBOX, EM_GETSEL, 0, 0L);
wStart = LOWORD(dwResult);
wEnd   = HIWORD(dwResult);
```

See Also EM_SETSEL

EM_GETWORDBREAKPROC

3.1

```
EM_GETWORDBREAKPROC
wParam = 0;      /* not used, must be zero */
lParam = 0L;     /* not used, must be zero */
```

An application sends the EM_GETWORDBREAKPROC message to an edit control to retrieve the current wordwrap function.

Parameters

This message has no parameters.

Return Value

The return value specifies the procedure-instance address of the application-defined wordwrap function. The return value is NULL if no wordwrap function exists.

Comments

A wordwrap function scans a text buffer (which contains text to be sent to the display), looking for the first word that does not fit on the current display line. The wordwrap function places this word at the beginning of the next line on the display. A wordwrap function defines at what point Windows should break a line of text for multiline edit controls, usually at a space character that separates two words.

See Also

EM_SETWORDBREAKPROC, **MakeProcInstance**, **WordBreakProc**

EM_LIMITTEXT

2.x

```
EM_LIMITTEXT
wParam = (WPARAM) cchMax;   /* text length           */
lParam = 0L;                /* not used, must be zero */
```

An application sends an EM_LIMITTEXT message to limit the length of the text the user can enter into an edit control.

Parameters

cchMax

Value of *wParam*. Specifies the length, in bytes, of the text the user can enter. If this parameter is zero, the text length is set to 65,535 bytes.

Return Value

This message does not return a value.

Comments

The EM_LIMITTEXT message limits only the text the user can enter. It has no effect on any text already in the edit control when the message is sent, nor does it affect the length of text copied to the edit control by the WM_SETTEXT message.

If an application uses the WM_SETTEXT message to place more text into an edit control than is specified in the EM_LIMITTEXT message, the user can edit the entire contents of the edit control.

See Also

WM_SETTEXT

EM_LINEFROMCHAR

2.x

```
EM_LINEFROMCHAR
wParam = (WPARAM) ich;     /* character index        */
lParam = 0L;               /* not used, must be zero */
```

An application sends an EM_LINEFROMCHAR message to retrieve the line number of the line that contains the specified character index. A character index is the number of characters from the beginning of the edit control.

This message is processed only by multiline edit controls.

Parameters

ich

Value of *wParam*. Specifies the character index of the character contained in the line whose number is to be retrieved. If the *ich* parameter is –1, either the line number of the current line (the line containing the caret) is retrieved or, if there is a selection, the line number of the line containing the beginning of the selection is retrieved.

Return Value

The return value is the zero-based line number of the line containing the character index specified by *ich*.

Example

This example sends an EM_LINEFROMCHAR message to retrieve the line number of the current line in a multiline edit control:

```
SendDlgItemMessage(hdlg, ID_MYEDITCONTROL,
    EM_LINEFROMCHAR, -1, 0L);
```

See Also

EM_LINEINDEX

EM_LINEINDEX

2.x

```
EM_LINEINDEX
wParam = (WPARAM) line;    /* line number           */
lParam = 0L;               /* not used, must be zero */
```

An application sends an EM_LINEINDEX message to retrieve the character index of a line within a multiline edit control. The character index is the number of characters from the beginning of the edit control to the specified line.

This message is processed only by multiline edit controls.

Parameters

line
Value of *wParam*. Specifies the zero-based line number. A value of –1 specifies the current line number (the line that contains the caret).

Return Value

The return value is the character index of the line specified in the *line* parameter, or it is –1 if the specified line number is greater than the number of lines in the edit control.

Example

This example uses the EM_GETLINECOUNT message to retrieve the number of lines in an edit control and then uses EM_LINEINDEX to retrieve the character index for the last line in the edit control:

```
WPARAM cLines, index;

cLines = (WPARAM) SendDlgItemMessage(hdlg, ID_MYEDITCONTROL,
    EM_GETLINECOUNT, 0, 0L);
index = (WPARAM) SendDlgItemMessage(hdlg, ID_MYEDITCONTROL,
    EM_LINEINDEX, cLines - 1, 0L);
```

See Also

EM_LINEFROMCHAR

EM_LINELENGTH

2.x

```
EM_LINELENGTH
wParam = (WPARAM) ich;     /* character index       */
lParam = 0L;               /* not used, must be zero */
```

An application sends an EM_LINELENGTH message to retrieve the length of a line in an edit control.

Parameters

ich

Value of *wParam*. Specifies the character index of a character in the line whose length is to be retrieved when EM_LINELENGTH is sent to a multiline edit control. If this parameter is −1, the message returns the number of unselected characters on lines containing selected characters. For example, if the selection extended from the fourth character of one line through the eighth character from the end of the next line, the return value would be 10 (three characters on the first line and seven on the next).

When EM_LINELENGTH is sent to a single-line edit control, this parameter is ignored.

Return Value

The return value is the length, in bytes, of the line specified by the *ich* parameter when an EM_LINELENGTH message is sent to a multiline edit control. The return value is the length, in bytes, of the text in the edit control when an EM_LINELENGTH message is sent to a single-line edit control.

Comments

Use the EM_LINEINDEX message to retrieve a character index for a given line number within a multiline edit control.

See Also

EM_LINEINDEX

EM_LINESCROLL

2.x

```
EM_LINESCROLL
wParam = 0;                        /* not used, must be zero         */
lParam = MAKELPARAM(dv, dh);       /* lines and characters to scroll */
```

An application sends an EM_LINESCROLL message to scroll the text of a multiline edit control.

This message is processed only by multiline edit controls.

Parameters

dv

Value of the low-order word of *lParam*. Specifies the number of lines to scroll vertically.

dh

Value of the high-order word of *lParam*. Specifies the number of character positions to scroll horizontally. This value is ignored if the edit control has either the ES_RIGHT or ES_CENTER style.

Return Value

The return value is nonzero if the message is sent to a multiline edit control, or it is zero if the message is sent to a single-line edit control.

Comments

The edit control does not scroll vertically past the last line of text in the edit control. If the current line plus the number of lines specified by the *dv* parameter exceeds the total number of lines in the edit control, the value is adjusted so that the last line of the edit control is scrolled to the top of the edit-control window.

The EM_LINESCROLL message can be used to scroll horizontally past the last character of any line.

Example

This example sends an EM_LINESCROLL message to scroll the text in a multiline edit control vertically by five lines:

```
SendDlgItemMessage(hdlg, ID_MYEDITCONTROL,
    EM_LINESCROLL, 0, MAKELONG(5, 0));
```

EM_REPLACESEL

2.x

```
EM_REPLACESEL
wParam = 0;                                  /* not used, must be zero */
lParam = (LPARAM) (LPCSTR) lpszReplace;      /* address of new string  */
```

An application sends an EM_REPLACESEL message to replace the current selection in an edit control with the text specified by the *lpszReplace* parameter.

Parameters

lpszReplace
Value of *lParam*. Points to a null-terminated string containing the replacement text.

Return Value

This message does not return a value.

Comments

Use the EM_REPLACESEL message when you want to replace only a portion of the text in an edit control. If you want to replace all of the text, use the WM_SETTEXT message.

If there is no current selection, the replacement text is inserted at the current cursor location.

Example

This example sets the selection to the beginning of the edit control and inserts the string "C:\":

```
SendDlgItemMessage(hdlg, ID_MYEDITCONTROL,
    EM_SETSEL, 0, MAKELONG(0, 0));
SendDlgItemMessage(hdlg, ID_MYEDITCONTROL,
    EM_REPLACESEL, 0, (LPARAM) ((LPCSTR) "C:\\"));
```

See Also

WM_SETTEXT

EM_SETHANDLE

2.x

```
EM_SETHANDLE
wParam = (WPARAM) (HLOCAL) hloc; /* handle of local memory object */
lParam = 0L;                     /* not used, must be zero        */
```

An application sends an EM_SETHANDLE message to set the handle to the local memory that will be used by a multiline edit control.

This message is processed only by multiline edit controls.

Parameters

hloc

Value of *wParam*. Identifies the local memory. This handle must have been created by a previous call to the **LocalAlloc** function using the LMEM_MOVEABLE flag. The memory should contain a null-terminated string, or the first byte of the allocated memory should be set to zero.

Return Value

This message does not return a value.

Comments

Before an application sets a new memory handle, it should send an EM_GETHANDLE message to retrieve the handle to the current memory buffer and should free that memory by using the **LocalFree** function.

Sending an EM_SETHANDLE message clears the undo buffer (EM_CANUNDO returns zero) and the internal modification flag (EM_GETMODIFY returns zero). The edit-control window is redrawn.

An application can send this message to a multiline edit control in a dialog box only if it has created the dialog box with the DS_LOCALEDIT style flag set.

Example

This example frees the current memory for the edit control, allocates new memory, and reads up to BUF_SIZE bytes of a file into the allocated memory. It then sends an EM_SETHANDLE message to set the handle of the edit control to the new memory, effectively placing up to BUF_SIZE bytes of the file into the edit control.

```
#define BUF_SIZE 4 * 1024

HANDLE hFile;
OFSTRUCT ofs;
HLOCAL hOldMem, hNewMem;
PSTR pBuf;
int cbRead;

/* Get the handle to the old memory and free it. */

hOldMem = (HLOCAL) SendDlgItemMessage(hdlg,
    ID_MYEDITCONTROL, EM_GETHANDLE, 0, 0L);
LocalFree(hOldMem);

/* Allocate new memory and read the file into it. */

hNewMem = LocalAlloc(LMEM_MOVEABLE, BUF_SIZE);
pBuf = LocalLock(hNewMem);
hFile = OpenFile("test.txt", &ofs, OF_READ);
cbRead = _lread(hFile, pBuf, BUF_SIZE);
pBuf[cbRead] = '\0';          /* terminating null character */
_lclose(hFile);

/* Adjust the buffer for the amount actually read in. */

LocalReAlloc(hNewMem, cbRead, 0);

/* Set the handle to the new buffer. */

LocalUnlock(hNewMem);
SendDlgItemMessage(hdlg, ID_MYEDITCONTROL,
    EM_SETHANDLE, hNewMem, 0L);
```

See Also

EM_CANUNDO, EM_GETHANDLE, EM_GETMODIFY, **LocalAlloc**, **LocalFree**

EM_SETMODIFY

2.x

```
EM_SETMODIFY
wParam = (WPARAM) (UINT) fModified;     /* modification flag      */
lParam = 0L;                            /* not used, must be zero */
```

An application sends an EM_SETMODIFY message to set or clear the modification flag for an edit control. The modification flag indicates whether the text within the edit control has been modified. It is automatically set whenever the user changes the text. An EM_GETMODIFY message can be sent to retrieve the value of the modification flag.

Parameters

fModified
Value of *wParam*. Specifies the new value for the modification flag. A value of TRUE indicates the text has been modified, and a value of FALSE indicates it has not been modified.

Return Value

This message does not return a value.

Example

This example sends an EM_SETMODIFY message to clear the modification flag:

```
SendDlgItemMessage(hdlg, ID_MYEDITCONTROL, EM_SETMODIFY, FALSE, 0L);
```

See Also

EM_GETMODIFY

EM_SETPASSWORDCHAR

3.0

```
EM_SETPASSWORDCHAR
wParam = (WPARAM) (UINT) ch;    /* character to display   */
lParam = 0L;                    /* not used, must be zero */
```

An application sends an EM_SETPASSWORDCHAR message to set or remove a password character displayed in an edit control when the user types text. When a password character is set, that character is displayed for each character the user types.

This message has no effect on a multiline edit control.

Parameters

ch
Value of *wParam*. Specifies the character to be displayed in place of the character typed by the user. If the *ch* parameter is zero, the actual characters typed by the user are displayed.

Return Value The return value is nonzero if the message is sent to an edit control.

Comments When the EM_SETPASSWORDCHAR message is received by an edit control, the edit control redraws all visible characters by using the character specified by the *ch* parameter.

If the edit control is created with the ES_PASSWORD style, the default password character is set to an asterisk (*). This style is removed if an EM_SETPASSWORDCHAR message is sent with the *wParam* parameter set to zero.

Example This example sends an EM_SETPASSWORDCHAR message to set the password character of an edit control to a question mark:

```
SendDlgItemMessage(hdlg, ID_MYEDITCONTROL,
    EM_SETPASSWORDCHAR, (WORD) '?', 0L);
```

See Also EM_GETPASSWORDCHAR

EM_SETREADONLY

3.1

```
EM_SETREADONLY
wParam = (WPARAM) (BOOL) fReadOnly;     /* read-only flag         */
lParam = 0L;                            /* not used, must be zero */
```

An application sends an EM_SETREADONLY message to set the read-only state of an edit control.

Parameters

fReadOnly
Value of *wParam*. Specifies whether to set or remove the read-only state of the edit control. A value of TRUE sets the state to read-only; a value of FALSE sets the state to read/write.

Return Value The return value is nonzero if the operation is successful, or it is zero if an error occurs.

Comments When the state of an edit control is set to read-only, the user cannot change the text within the edit control.

Example This example sets the state of an edit control to read-only:

```
SendDlgItemMessage(hdlg, IDD_EDIT, EM_SETREADONLY,
    TRUE, 0L);
```

EM_SETRECT

2.x

```
EM_SETRECT
wParam = 0;                              /* not used, must be zero */
lParam = (LPARAM) (const RECT FAR*) lprc; /* address of RECT    */
```

An application sends an EM_SETRECT message to set the formatting rectangle of a multiline edit control. The formatting rectangle is the limiting rectangle of the text. The limiting rectangle is independent of the size of the edit-control window. When the edit control is first created, the formatting rectangle is the same as the client area of the edit-control window. By using the EM_SETRECT message, an application can make the formatting rectangle larger or smaller than the edit-control window.

This message is processed only by multiline edit controls.

Parameters

lprc

Value of *lParam*. Points to a **RECT** structure that specifies the new dimensions of the rectangle. The **RECT** structure has the following form:

```
typedef struct tagRECT {     /* rc */
   int left;
   int top;
   int right;
   int bottom;
} RECT;
```

Return Value

This message does not return a value.

Comments

The EM_SETRECT message causes the text of the edit control to be redrawn. To change the size of the formatting rectangle without redrawing the text, use the EM_SETRECTNP message.

If the edit control does not have a horizontal scroll bar, and the formatting rectangle is set to be larger than the edit-control window, lines of text exceeding the width of the edit-control window (but smaller than the width of the formatting rectangle) are clipped instead of wrapped.

If the edit control contains a border, the formatting rectangle is reduced by the size of the border. If you are adjusting the rectangle returned by an EM_GETRECT message, you must remove the size of the border before using the rectangle with the EM_SETRECT message.

Example

This example retrieves the current formatting rectangle for a multiline edit control, removes the border width dimensions, and sets the right border to 32767 so that all text sent to the edit control is clipped rather than wrapped if it exceeds the width of the edit-control window. The example then sends an EM_SETRECT message to set the new formatting rectangle.

```
RECT rect;

SendDlgItemMessage(hdlg, ID_MYEDITCONTROL,
    EM_GETRECT, 0, (LPARAM) (RECT FAR*) &rect);
rect.left = 0;              /* remove border width  */
rect.right = 32767;         /* clip all lines       */
rect.bottom += rect.top;    /* remove border height */
rect.top = 0;               /* remove border height */
SendDlgItemMessage(hdlg, ID_MYEDITCONTROL,
    EM_SETRECT, 0, (LPARAM) (RECT FAR*) &rect);
```

See Also

EM_GETRECT, EM_SETRECTNP, **MoveWindow**

EM_SETRECTNP

2.x

```
EM_SETRECTNP
wParam = 0;                              /* not used, must be zero    */
lParam = (LPARAM) (const RECT FAR*) lprc; /* address of RECT          */
```

An application sends an EM_SETRECTNP message to set the formatting rectangle of a multiline edit control. The formatting rectangle is the limiting rectangle of the text. The limiting rectangle is independent of the size of the edit-control window. When the edit control is first created, the formatting rectangle is the same as the client area of the edit-control window. By using the EM_SETRECTNP message, an application can make the formatting rectangle larger or smaller than the edit-control window.

The EM_SETRECTNP message is identical to the EM_SETRECT message, except that the edit-control window is not redrawn.

This message is processed only by multiline edit controls.

Parameters

lprc
Value of *lParam*. Points to a **RECT** structure that specifies the new dimensions of the rectangle. The **RECT** structure has the following form:

```
typedef struct tagRECT {    /* rc */
   int left;
   int top;
   int right;
   int bottom;
} RECT;
```

Return Value

This message does not return a value.

See Also

EM_SETRECT

EM_SETSEL

2.x

```
EM_SETSEL
wParam = (WPARAM) (UINT) fScroll;       /* flag for caret scrolling */
lParam = MAKELPARAM(ichStart, ichEnd);  /* start and end positions  */
```

An application sends an EM_SETSEL message to select a range of characters in an edit control.

Parameters

fScroll
Value of *wParam*. When this parameter is zero, the caret is scrolled into view. When this parameter is 1, the caret is not scrolled into view.

ichStart
Value of the low-order word of *lParam*. Specifies the starting position.

ichEnd
Value of the high-order word of *lParam*. Specifies the ending position.

Return Value

The return value is nonzero if the message is sent to an edit control.

Comments

If the *ichStart* parameter is 0 and the *ichEnd* parameter is −1, all the text in the edit control is selected. If *ichStart* is −1, any current selection is removed. The caret is placed at the end of the selection indicated by the greater of the two values *ichEnd* and *ichStart*.

Example

This example sends an EM_SETSEL message to select the entire contents of an edit control. It then sends a WM_CUT message to copy the contents of the edit control to the clipboard and then to delete the contents of the edit control.

```
SendDlgItemMessage(hdlg, ID_MYEDITCONTROL,
    EM_SETSEL, 0, MAKELONG(0, -1));
SendDlgItemMessage(hdlg, ID_MYEDITCONTROL,
    WM_CUT, 0, 0L);
```

See Also

EM_GETSEL, EM_REPLACESEL

EM_SETTABSTOPS

3.0

```
EM_SETTABSTOPS
wParam = (WPARAM) cTabs;                          /* number of tab stops */
lParam = (LPARAM) (const int FAR*) lpTabs;        /* tab-stop array      */
```

An application sends an EM_SETTABSTOPS message to set the tab stops in a multiline edit control (MLE). When text is copied to an MLE, any tab character in the text causes space to be generated up to the next tab stop.

This message is processed only by MLEs.

Parameters

cTabs

Value of *wParam*. Specifies the number of tab stops contained in the *lpTabs* parameter. If this parameter is 0, the *lpTabs* parameter is ignored and default tab stops are set at every 32 dialog box units. If this parameter is 1, tab stops are set at every *n* dialog box units, where *n* is the distance pointed to by the *lpTabs* parameter. If the *cTabs* parameter is greater than 1, *lpTabs* points to an array of tab stops.

lpTabs

Low and high-order words of *lParam*. Points to an array of unsigned integers specifying the tab stops, in dialog box units. If the *cTabs* parameter is 1, *lpTabs* points to an unsigned integer containing the distance between all tab stops, in dialog units.

Return Value

The return value is nonzero if the tabs were set; otherwise, the return value is zero.

Comments

The EM_SETTABSTOPS message does not automatically redraw the edit-control window. If the application is changing the tab stops for text already in the edit control, it should call the **InvalidateRect** function to redraw the edit-control window.

Example

This example sends an EM_SETTABSTOPS message to set tab stops at every 64 dialog box units. It then calls **InvalidateRect** to redraw the edit-control window.

```
WORD wTabSpacing = 64;

SendDlgItemMessage(hdlg, ID_MYEDITCONTROL,
    EM_SETTABSTOPS, 1, (LPARAM) (int far*) &wTabSpacing);
InvalidateRect(GetDlgItem(hdlg, ID_MYEDITCONTROL),
    NULL, TRUE);
```

See Also

GetDialogBaseUnits

EM_SETWORDBREAKPROC

3.1

```
EM_SETWORDBREAKPROC
wParam = 0;                                    /* not used, must be zero */
lParam = (LPARAM) (EDITWORDBREAKPROC) ewbprc; /* address of function */
```

An application sends the EM_SETWORDBREAKPROC message to an edit control to replace the default wordwrap function with an application-defined wordwrap function.

Parameters

ewbprc

Value of *lParam*. Specifies the procedure-instance address of the application-defined wordwrap function. The **MakeProcInstance** function must be used to create the address. For more information, see the description of the **WordBreakProc** callback function.

Return Value

This message does not return a value.

Comments

A wordwrap function scans a text buffer (which contains text to be sent to the display), looking for the first word that does not fit on the current display line. The wordwrap function places this word at the beginning of the next display line.

A wordwrap function defines the point at which Windows should break a line of text for multiline edit controls, usually at a space character that separates two words. Either a multiline or a single-line edit control might call this function when the user presses arrow keys in combination with the CTRL key to move the cursor to the next word or previous word. The default wordwrap function breaks a line of text at a space character. The application-defined function may define wordwrap to occur at a hyphen or a character other than the space character.

See Also

EM_GETWORDBREAKPROC, **MakeProcInstance**, **WordBreakProc**

EM_UNDO

2.x

```
EM_UNDO
wParam = 0;      /* not used, must be zero */
lParam = 0L;     /* not used, must be zero */
```

An application sends an EM_UNDO message to undo the last edit-control operation.

Parameters

This message has no parameters.

Return Value

The return value is always nonzero for a single-line edit control. For a multiline edit control, the return value is nonzero if the undo operation is successful or zero if the undo operation fails.

Comments

An undo operation can also be undone. For example, you can restore deleted text with the first EM_UNDO message and remove the text again with a second EM_UNDO message as long as there is no intervening edit-control operation.

Example

This example undoes the last edit-control operation:

```
SendDlgItemMessage(hdlg, ID_MYEDITCONTROL, EM_UNDO, 0, 0L);
```

See Also

EM_CANUNDO

LB_ADDSTRING

2.x

```
LB_ADDSTRING
wParam = 0;                          /* not used, must be zero   */
lParam = (LPARAM) (LPCSTR) lpsz; /* address of string to add */
```

An application sends an LB_ADDSTRING message to add a string to a list box. If the list box does not have the CBS_SORT style, the string is added to the end of the list. Otherwise, the string is inserted into the list and the list is sorted.

Parameters

lpsz

Value of *lParam*. Points to the null-terminated string that is to be added. If the list box was created with an owner-drawn style but without the LBS_HASSTRINGS style, the value of the *lpsz* parameter is stored rather than the string it would otherwise point to.

Return Value

The return value is the zero-based index to the string in the list box. The return value is LB_ERR if an error occurs; the return value is LB_ERRSPACE if insufficient space is available to store the new string.

Comments

If an owner-drawn list box was created with the LBS_SORT style but not the LBS_HASSTRINGS style, the WM_COMPAREITEM message is sent one or more times to the owner of the list box so the new item can be properly placed in the list box.

Example

This example adds the string "my string" to a list box:

```
DWORD dwIndex;

dwIndex = SendDlgItemMessage(hdlg, ID_MYLISTBOX,
    LB_ADDSTRING, 0, (LPARAM) ((LPCSTR) "my string"));
```

See Also

LB_DELETESTRING, LB_INSERTSTRING, WM_COMPAREITEM

LB_DELETESTRING

2.x

```
LB_DELETESTRING
wParam = (WPARAM) index;     /* index of string to delete */
lParam = 0L;                 /* not used, must be zero    */
```

An application sends an LB_DELETESTRING message to delete a string in a list box.

Parameters

index
Value of *wParam*. Specifies the zero-based index of the string to delete.

Return Value

The return value is a count of the strings remaining in the list. The return value is LB_ERR if the *index* parameter specifies an index greater than the number of items in the list.

Comments

If the list box was created with an owner-drawn style but without the LBS_HASSTRINGS style, a WM_DELETEITEM message is sent to the owner of the list box so that the application can free any additional data associated with the item.

Example

This example deletes the first string in a list box:

```
DWORD dwRemaining;

dwRemaining = SendDlgItemMessage(hdlg, ID_MYLISTBOX,
    LB_DELETESTRING, 0, 0L);
```

See Also

LB_ADDSTRING, WM_DELETEITEM

LB_DIR

2.x

```
LB_DIR
wParam = (WPARAM) (UINT) uAttrs;         /* file attributes          */
lParam = (LPARAM) (LPCSTR) lpszFileSpec; /* filename string's address */
```

An application sends an LB_DIR message to add a list of filenames to a list box.

Parameters

uAttrs

Value of *wParam*. Specifies the attributes of the files to be added to the list box. It can be any combination of the following values:

Value	Meaning
0x0000	File can be read from or written to.
0x0001	File can be read from but not written to.
0x0002	File is hidden and does not appear in a directory listing.
0x0004	File is a system file.
0x0010	The name pointed to by the *lpszFileSpec* parameter specifies a directory.
0x0020	File has been archived.
0x4000	All drives that match the name specified by the *lpszFileSpec* parameter are included.
0x8000	Exclusive flag. If the exclusive flag is set, only files of the specified type are listed. Otherwise, files of the specified type are listed in addition to files that do not match the specified type.

lpszFileSpec

Value of *lParam*. Points to the null-terminated string that specifies the filename to add to the list. If the filename contains wildcards (for example, *.*), all files that match and have the attributes specified by the *uAttrs* parameter are added to the list.

Return Value

The return value is the zero-based index of the last filename added to the list. The return value is LB_ERR if an error occurs; the return value is LB_ERRSPACE if insufficient space is available to store the new strings.

Example

This example adds the names of all available drives to a list box:

```
DWORD dwIndexLastItem;

dwIndexLastItem = SendDlgItemMessage(hdlg, ID_MYLISTBOX, LB_DIR,
    0x4000 | 0x8000, (LPARAM) ((LPCSTR) "*"));
```

See Also

DlgDirList

LB_FINDSTRING

3.0

```
LB_FINDSTRING
wParam = (WPARAM) indexStart;        /* item before start of search */
lParam = (LPARAM) (LPCSTR) lpszFind; /* address of search string    */
```

An application sends an LB_FINDSTRING message to find the first string in a list box that contains the specified prefix.

Parameters

indexStart

Value of *wParam*. Specifies the zero-based index of the item before the first item to be searched. When the search reaches the bottom of the list box, it continues from the top of the list box back to the item specified by the *indexStart* parameter. If *indexStart* is –1, the entire list box is searched from the beginning.

lpszFind

Value of *lParam*. Points to the null-terminated string that contains the prefix to search for. The search is not case-sensitive, so this string can contain any combination of uppercase and lowercase letters.

Return Value

The return value is the index of the matching item, or it is LB_ERR if the search was unsuccessful.

Comments

If the list box was created with an owner-drawn style but without the LBS_HASSTRINGS style, the action taken by LB_FINDSTRING depends on whether the LBS_SORT style is used. If LBS_SORT is used, WM_COMPAREITEM messages are sent to the owner of the list box to determine which item matches the specified string. Otherwise, LB_FINDSTRING attempts to match the doubleword value against the value of *lpszFind*.

Example

This example searches for the string "my string" in a list box and copies it, if found, to the szBuf buffer:

```
char szBuf[20];
DWORD dwIndex;

dwIndex = SendDlgItemMessage(hdlg, ID_MYLISTBOX,
    LB_FINDSTRING, 0, (LPARAM) ((LPCSTR) "my string"));
if (dwIndex != LB_ERR)
    SendDlgItemMessage(hdlg, ID_MYLISTBOX,
        LB_GETTEXT, (WPARAM) dwIndex, (LPARAM) ((LPCSTR) szBuf));
```

See Also

LB_ADDSTRING, LB_FINDSTRINGEXACT, LB_INSERTSTRING

LB_FINDSTRINGEXACT

3.1

```
LB_FINDSTRINGEXACT
wParam = (WPARAM) indexStart;        /* item before start of search */
lParam = (LPARAM) (LPCSTR) lpszFind; /* address of search string    */
```

An application sends an LB_FINDSTRINGEXACT message to find the first list box string that matches the string specified in the *lpszFind* parameter.

Parameters

indexStart

Value of *wParam*. Specifies the zero-based index of the item before the first item to be searched. When the search reaches the bottom of the list box, it continues from the top of the list box back to the item specified by the *indexStart* parameter. If *indexStart* is –1, the entire list box is searched from the beginning.

lpszFind

Value of *lParam*. Points to the null-terminated string to search for. This string can contain a complete filename, including the extension. The search is not case-sensitive, so the string can contain any combination of uppercase and lowercase letters.

Return Value

The return value is the index of the matching item, or it is LB_ERR if the search was unsuccessful.

Comments

If the list box was created with an owner-drawn style but without the LBS_HASSTRINGS style, the action taken by LB_FINDSTRINGEXACT depends on whether the LBS_SORT style is used. If LBS_SORT is used, WM_COMPAREITEM messages are sent to the owner of the list box to determine which item matches the specified string. Otherwise, LB_FINDSTRINGEXACT attempts to match the doubleword value against the value of *lpszFind.*

See Also

LB_ADDSTRING, LB_FINDSTRING, LB_INSERTSTRING

LB_GETCARETINDEX

3.1

```
LB_GETCARETINDEX
wParam = 0;      /* not used, must be zero */
lParam = 0L;     /* not used, must be zero */
```

An application sends an LB_GETCARETINDEX message to determine the index of the item that has the focus rectangle in a multiple-selection list box. The item may or may not be selected.

Parameters

This message has no parameters.

Return Value

The return value is the zero-based index of the item that has the focus rectangle in a list box. If the list box is a single-selection list box, the return value is the index of the item that is selected, if any.

Example

This example sends an LB_GETCARETINDEX message to retrieve the index of the item that has the focus rectangle in the list box:

```
LRESULT lrIndex;

lrIndex = SendDlgItemMessage(hdlg, ID_MYLISTBOX,
    LB_GETCARETINDEX, 0, 0L);
```

See Also

LB_SETCARETINDEX

LB_GETCOUNT

2.x

```
LB_GETCOUNT
wParam = 0;      /* not used, must be zero */
lParam = 0L;     /* not used, must be zero */
```

An application sends an LB_GETCOUNT message to retrieve the number of items in a list box.

Parameters

This message has no parameters.

Return Value

The return value is the number of items in the list box, or it is LB_ERR if an error occurs.

Comments

The returned count is one greater than the index value of the last item (the index is zero-based).

Example

This example retrieves the number of items in a list box:

```
DWORD cListItems;

cListItems = SendDlgItemMessage(hdlg, ID_MYLISTBOX, LB_GETCOUNT, 0, 0);
```

LB_GETCURSEL

2.x

```
LB_GETCURSEL
wParam = 0;      /* not used, must be zero */
lParam = 0L;     /* not used, must be zero */
```

An application sends an LB_GETCURSEL message to retrieve the index of the currently selected item, if any, in a single-selection list box.

Parameters

This message has no parameters.

Return Value

The return value is the zero-based index of the currently selected item. It is LB_ERR if no item is currently selected.

Comments

An application should use the LB_GETCARETINDEX to retrieve the index of the item that has the focus rectangle in a multiple-selection list box.

The LB_GETCURSEL message cannot be sent to a multiple-selection list box.

Example

This example retrieves the index of the currently selected string in a list box and then retrieves that string:

```
char szBuf[20];
DWORD dwIndex;

dwIndex = SendDlgItemMessage(hdlg, ID_MYLISTBOX, LB_GETCURSEL, 0, 0);
if (dwIndex != LB_ERR)
    SendDlgItemMessage(hdlg, ID_MYLISTBOX,
        LB_GETTEXT, (WPARAM) dwIndex, (LPARAM) ((LPCSTR) szBuf));
```

See Also

LB_GETCARETINDEX

LB_GETHORIZONTALEXTENT

3.0

```
LB_GETHORIZONTALEXTENT
wParam = 0;      /* not used, must be zero */
lParam = 0L;     /* not used, must be zero */
```

An application sends the LB_GETHORIZONTALEXTENT message to retrieve from a list box the width, in pixels, by which the list box can be scrolled horizontally if the list box has a horizontal scroll bar.

Parameters

This message has no parameters.

Return Value

The return value is the scrollable width of the list box, in pixels.

Comments

To respond to the LB_GETHORIZONTALEXTENT message, the list box must have been defined with the WS_HSCROLL style.

Example

This example gets the horizontal extent of a list box:

```
SendDlgItemMessage(hDlg, ID_MYLISTBOX,
    LB_GETHORIZONTALEXTENT, 0, 0L);
```

See Also

LB_SETHORIZONTALEXTENT

LB_GETITEMDATA

3.0

```
LB_GETITEMDATA
wParam = (WPARAM) index;     /* item index             */
lParam = 0L;                 /* not used, must be zero */
```

An application sends the LB_GETITEMDATA message to retrieve the application-supplied doubleword value associated with the specified item in a list box. (This is the value of the *lParam* parameter of an LB_SETITEMDATA message.)

Parameters

index
Value of *wParam*. Specifies the zero-based index of the item.

Return Value

The return value is the doubleword value associated with the item, or it is LB_ERR if an error occurs.

Example

This example retrieves the value associated with an item in a list box. The value is the handle of a global memory object.

```
HGLOBAL hLBData;
LPSTR   lpLBData;
HWND    hListBox;
WPARAM  nIndex;

if ((hLBData = LOWORD(SendMessage(hListBox, LB_GETITEMDATA,
        nIndex, 0L)))) {
    if ((lpLBData = GlobalLock(hLBData))) {
        .
        . /* Access or manipulate the data */
        .

        GlobalUnlock(hLBData);
    }
}
```

See Also

LB_ADDSTRING, LB_INSERTSTRING, LB_SETITEMDATA

LB_GETITEMHEIGHT

3.1

```
LB_GETITEMHEIGHT
wParam = (WPARAM) index;     /* item index             */
lParam = 0L;                 /* not used, must be zero */
```

An application sends an LB_GETITEMHEIGHT message to determine the height of items in a list box.

Parameters

index

Value of *wParam*. Specifies the zero-based index of the item in the list box. This parameter is used only if the list box has the LBS_OWNERDRAWVARIABLE style; otherwise, it should be set to zero.

Return Value

The return value is the height, in pixels, of the items in the list box. The return value is the height of the item specified by the *index* parameter if the list box has the LBS_OWNERDRAWVARIABLE style. The return value is LB_ERR if an error occurs.

Example

This example sends LB_GETITEMHEIGHT to retrieve the height of the items in a list box:

```
LRESULT lrHeight;

lrHeight = SendDlgItemMessage(hdlg, ID_MYLISTBOX,
    LB_GETITEMHEIGHT, 0, 0L);
```

See Also

LB_SETITEMHEIGHT

LB_GETITEMRECT

3.0

```
LB_GETITEMRECT
wParam = (WPARAM) index;            /* item index                */
lParam = (LPARAM) (RECT FAR*) lprc; /* address of RECT structure */
```

An application sends an LB_GETITEMRECT message to retrieve the dimensions of the rectangle that bounds an item as it is currently displayed in the list box window.

Parameters

index
Value of *wParam*. Specifies the zero-based index of the item.

lprc
Value of *lParam*. Specifies a long pointer to a **RECT** structure that receives the client coordinates for the item in the list box. The **RECT** structure has the following form:

```
typedef struct tagRECT {    /* rc */
   int left;
   int top;
   int right;
   int bottom;
} RECT;
```

Return Value

The return value is LB_ERR if an error occurs.

LB_GETSEL

2.x

```
LB_GETSEL
wParam = (WPARAM) index;    /* item index             */
lParam = 0L;                /* not used, must be zero */
```

An application sends an LB_GETSEL message to retrieve the selection state of an item.

Parameters

index
Value of *wParam*. Specifies the zero-based index of the item.

Return Value

The return value is a positive number if an item is selected; otherwise, it is zero. The return value is LB_ERR if an error occurs.

See Also

LB_SETSEL

LB_GETSELCOUNT

3.0

```
LB_GETSELCOUNT
wParam = 0;      /* not used, must be zero */
lParam = 0L;     /* not used, must be zero */
```

An application sends an LB_GETSELCOUNT message to retrieve the total number of selected items in a multiple-selection list box.

Parameters

This message has no parameters.

Return Value

The return value is the count of selected items in a list box. The return value is LB_ERR if the list box is a single-selection list box.

See Also

LB_SETSEL

LB_GETSELITEMS

3.0

```
LB_GETSELITEMS
wParam = (WPARAM) cItems;             /* maximum number of items */
lParam = (LPARAM) (int FAR*) lpItems; /* address of buffer       */
```

An application sends an LB_GETSELITEMS message to fill a buffer with an array of integers that specify the item numbers of selected items in a multiple-selection list box.

Parameters

cItems

Value of *wParam*. Specifies the maximum number of selected items whose item numbers are to be placed in the buffer.

lpItems

Value of *lParam*. Specifies a long pointer to a buffer large enough for the number of integers specified by the *cItems* parameter.

Return Value

The return value is the actual number of items placed in the buffer. The return value is LB_ERR if the list box is a single-selection list box.

See Also

LB_GETSELCOUNT

LB_GETTEXT

2.x

```
LB_GETTEXT
wParam = (WPARAM) index;              /* item index        */
lParam = (LPARAM) (LPCSTR) lpszBuffer; /* address of buffer */
```

An application sends an LB_GETTEXT message to retrieve a string from a list box.

Parameters

index
Value of *wParam*. Specifies the zero-based index of the string to retrieve.

lpszBuffer
Value of *lParam*. Points to the buffer that receives the string. The buffer must have sufficient space for the string and a terminating null character. An LB_GETTEXTLEN message can be sent before the LB_GETTEXT message to retrieve the length, in bytes, of the string.

Return Value

The return value is the length of the string, in bytes, excluding the terminating null character. The return value is LB_ERR if the *index* parameter does not specify a valid index.

Comments

If the list box was created with an owner-drawn style but without the LBS_HASSTRINGS style, the buffer pointed to by the *lpszBuffer* parameter receives the doubleword value associated with the item.

Example

This example retrieves the length of the first item in the list box, allocates sufficient memory for the string, and then sends an LB_GETTEXT message to retrieve the string:

```
DWORD cbItemString;
PSTR psz;

cbItemString = SendDlgItemMessage(hdlg, ID_MYLISTBOX,
    LB_GETTEXTLEN, 0, 0L);
if (cbItemString != LB_ERR) {
    psz = (PSTR) LocalAlloc(LMEM_FIXED, (WORD) cbItemString);
    SendDlgItemMessage(hdlg, ID_MYLISTBOX,
        LB_GETTEXT, 0, (LPARAM) ((LPCSTR) psz));
}
```

See Also

LB_GETTEXTLEN

LB_GETTEXTLEN

2.x

```
LB_GETTEXTLEN
wParam = (WPARAM) index;     /* item index             */
lParam = 0L;                 /* not used, must be zero */
```

An application sends an LB_GETTEXTLEN message to retrieve the length of a string in a list box.

Parameters

index
Value of *wParam*. Specifies the zero-based index of the string.

Return Value

The return value is the length of the string, in bytes, excluding the terminating null character. The return value is LB_ERR if the *index* parameter does not specify a valid index.

Example

This example retrieves the length of the first item in the list box:

```
DWORD cbItemString;

cbItemString = SendDlgItemMessage(hdlg, ID_MYLISTBOX,
    LB_GETTEXTLEN, 0, 0L);
```

See Also

LB_GETTEXT

LB_GETTOPINDEX

3.0

```
LB_GETTOPINDEX
wParam = 0;      /* not used, must be zero */
lParam = 0L;     /* not used, must be zero */
```

An application sends an LB_GETTOPINDEX message to retrieve the index of the first visible item in a list box. Initially, the item with index 0 is at the top of the list box, but if the list box is scrolled, another item may be at the top.

Parameters

This message has no parameters.

Return Value

The return value is the zero-based index of the first visible item in a list box.

See Also

LB_SETTOPINDEX

LB_INSERTSTRING

2.x

```
LB_INSERTSTRING
wParam = (WPARAM) index;          /* item index                  */
lParam = (LPARAM) (LPCSTR) lpsz;  /* address of string to insert */
```

An application sends an LB_INSERTSTRING message to insert a string into a list box. Unlike the LB_ADDSTRING message, the LB_INSERTSTRING message does not cause a list with the LBS_SORT style to be sorted.

Parameters

index
Value of *wParam*. Specifies the zero-based index of the position at which to insert the string. If this parameter is –1, the string is added to the end of the list.

lpsz
Value of *lParam*. Points to the null-terminated string that is to be inserted. If the list was created with an owner-drawn style but without the LBS_HASSTRINGS style, the value of the *lpsz* parameter is stored rather than the string it would otherwise point to.

Return Value

The return value is the index of the position at which the string was inserted. The return value is LB_ERR if an error occurs. The return value is LB_ERRSPACE if insufficient space is available to store the new string.

Example

This example inserts the string "my string" into the third position of the list box:

```
SendDlgItemMessage(hdlg, ID_MYLISTBOX,
    LB_INSERTSTRING, 2, (LPARAM) ((LPCSTR) "my string"));
```

See Also

LB_ADDSTRING

LB_RESETCONTENT

2.x

```
LB_RESETCONTENT
wParam = 0;     /* not used, must be zero */
lParam = 0L;    /* not used, must be zero */
```

An application sends an LB_RESETCONTENT message to remove all items from a list box.

Parameters

This message has no parameters.

Return Value

This message does not return a value.

Comments

If the list box was created with an owner-drawn style but without the LBS_HASSTRINGS style, the owner of the list box receives a WM_DELETEITEM message for each item in the list box.

Example

This example removes all items from a list box:

```
SendDlgItemMessage(hdlg, ID_MYLISTBOX, LB_RESETCONTENT, 0, 0L);
```

See Also

WM_DELETEITEM

LB_SELECTSTRING

2.x

```
LB_SELECTSTRING
wParam = (WPARAM) indexStart;        /* item before start of search */
lParam = (LPARAM) (LPCSTR) lpszFind; /* address of search string    */
```

An application sends an LB_SELECTSTRING message to search the list box for an item that matches the specified string, and if a matching item is found, to select the item.

Parameters

indexStart

Value of *wParam*. Specifies the zero-based index of the item before the first item to be searched. When the search reaches the bottom of the list box, it continues from the top of the list box back to the item specified by the *indexStart* parameter. If *indexStart* is –1, the entire list box is searched from the beginning.

lpszFind

Value of *lParam*. Points to the null-terminated string that contains the prefix to search for. The search is not case-sensitive, so this string can contain any combination of uppercase and lowercase letters.

Return Value

The return value is the index of the selected item if the search was successful. The return value is LB_ERR if the search was unsuccessful and the current selection is not changed.

Comments

The list box is scrolled, if necessary, to bring the selected item into view.

An item is selected only if its initial characters (from the starting point) match the characters in the string specified by the *lpszFind* parameter.

If the list box was created with an owner-drawn style but without the LBS_HASSTRINGS style, the action taken by LB_SELECTSTRING depends on whether the LBS_SORT style is used. If LBS_SORT is used, WM_COMPAREITEM messages are sent to the owner of the list box to determine which item matches the specified string. Otherwise, LB_SELECTSTRING attempts to match the doubleword value against the value of *lpszFind.*

Example

This example searches the entire list box for an item that matches the string "my string" and, if the item is found, selects it:

```
DWORD dwIndexFoundString;

dwIndexFoundString = SendDlgItemMessage(hdlg, ID_MYLISTBOX,
    LB_SELECTSTRING, -1, (LPARAM) ((LPCSTR) "my string"));
```

See Also

LB_ADDSTRING, LB_FINDSTRING, LB_INSERTSTRING

LB_SELITEMRANGE

3.0

```
LB_SELITEMRANGE
wParam = (WPARAM) (BOOL) fSelect;   /* selection flag       */
lParam = MAKELPARAM(wFirst, wLast); /* first and last items */
```

An application sends an LB_SELITEMRANGE message to select one or more consecutive items in a multiple-selection list box.

Parameters

fSelect
Value of *wParam.* Specifies how to set the selection. If the *fSelect* parameter is nonzero, the string is selected and highlighted; if *fSelect* is zero, the highlight is removed and the string is no longer selected.

wFirst
Value of the low-order word of *lParam.* Specifies the zero-based index of the first item to set.

wLast
Value of the high-order word of *lParam.* Specifies the zero-based index of the last item to set.

Return Value

The return value is LB_ERR if an error occurs.

Comments

This message should be used only with multiple-selection list boxes.

LB_SETCARETINDEX

3.1

```
LB_SETCARETINDEX
wParam = (WPARAM) index;          /* item index              */
lParam = MAKELPARAM(fScroll, 0); /* flag for scrolling item */
```

An application sends an LB_SETCARETINDEX message to set the focus rectangle to the item at the specified index in a multiple-selection list box. If the item is not visible, it is scrolled into view.

Parameters

index
Value of *wParam*. Specifies the zero-based index of the item to receive the focus rectangle in the list box.

fScroll
Value of *lParam*. If this value is zero, the item is scrolled until it is fully visible. If this value is nonzero, the item is scrolled until it is at least partially visible.

Return Value

The return value is LB_ERR if an error occurs.

Example

This example sends an LB_SETCARETINDEX message to set the focus rectangle to an item in a list box:

```
WPARAM wIndex;

wIndex = 0;      /* set index to first item */

SendDlgItemMessage(hdlg, ID_MYLISTBOX, LB_SETCARETINDEX,
    wIndex, 0L);
```

See Also

LB_GETCARETINDEX

LB_SETCOLUMNWIDTH

3.0

```
LB_SETCOLUMNWIDTH
wParam = (WPARAM) cxColumn;      /* column width          */
lParam = 0L;                     /* not used, must be zero */
```

An application sends an LB_SETCOLUMNWIDTH message to a multiple-column list box (created with the LBS_MULTICOLUMN style) to set the width, in pixels, of all columns in the list box.

Parameters

cxColumn
Value of *wParam*. Specifies the width, in pixels, of all columns.

Return Value

This message does not return a value.

Example

This example sets the width of the columns in a multiple-column list box:

```
WPARAM wColWidth;

wColWidth = 100;     /* set column width to 100 pixels */

SendDlgItemMessage(hDlg, ID_MYLISTBOX, LB_SETCOLUMNWIDTH,
    wColWidth, 0L);
```

LB_SETCURSEL

2.x

```
LB_SETCURSEL
wParam = (WPARAM) index;     /* item index              */
lParam = 0L;                 /* not used, must be zero */
```

An application sends an LB_SETCURSEL message to select a string and scroll it into view, if necessary. When the new string is selected, the list box removes the highlight from the previously selected string.

Parameters

index
Value of *wParam*. Specifies the zero-based index of the string that is selected. If the *index* parameter is –1, the list box is set to have no selection.

Return Value

The return value is LB_ERR if an error occurs. The return value will be LB_ERR even though no error has occurred if the *index* parameter is –1.

Comments

This message should be used only with single-selection list boxes. It cannot be used to set or remove a selection in a multiple-selection list box.

See Also

LB_GETCURSEL

LB_SETHORIZONTALEXTENT

3.0

```
LB_SETHORIZONTALEXTENT
wParam = (WPARAM) cxExtent; /* horizontal scroll width */
lParam = 0L;                /* not used, must be zero  */
```

An application sends the LB_SETHORIZONTALEXTENT message to set the width, in pixels, by which a list box can be scrolled horizontally. If the size of the list box is smaller than this value, the horizontal scroll bar horizontally scrolls items in the list box. If the size of the list box is equal to or greater than this value, the horizontal scroll bar is hidden.

Parameters

cxExtent
Value of *wParam*. Specifies the number of pixels by which the list box can be scrolled.

Return Value

This message does not return a value.

Comments

To respond to the LB_SETHORIZONTALEXTENT message, the list box must have been defined with the WS_HSCROLL style.

By default, the horizontal extent of a list box is zero. Windows does not display the scroll bar unless the horizontal extent is set to a value greater than the width, in pixels, of the client area of the list box.

Example

This example sets the horizontal extent of a list box based on the width of the string about to be added to the list box. The horizontal extent is set if the string is wider than the widest string in the list box and is wider than the client area of the list box.

```
DWORD dwStringExt;
HDC hdcLB;
PSTR pszString;
TEXTMETRIC tm;
WORD wLongest;
WORD wLBWidth;

dwStringExt = GetTextExtent(hdcLB, (LPSTR) pszString,
    strlen(pszString)) + tm.tmAveCharWidth;
```

```
if ((LOWORD(dwStringExt) > wLongest) &&
        (LOWORD(dwStringExt) > wLBWidth)) {
    SendDlgItemMessage(hDlg, ID_MYLISTBOX, LB_SETHORIZONTALEXTENT,
        LOWORD(dwStringExt), 0L);
    wLongest = LOWORD(dwStringExt);
}

SendDlgItemMessage(hDlg, ID_MYLISTBOX, LB_ADDSTRING, 0,
    (LPARAM) ((LPCSTR) pszString));
```

See Also LB_GETHORIZONTALEXTENT

LB_SETITEMDATA

3.0

```
LB_SETITEMDATA
wParam = (WPARAM) index;    /* item index                    */
lParam = (LPARAM) dwData;   /* value to associate with item */
```

An application sends the LB_SETITEMDATA message to set a doubleword value associated with the specified item in a list box.

Parameters

index
Value of *wParam*. Specifies the zero-based index of the item.

dwData
Value of *lParam*. Specifies the value to be associated with the item.

Return Value The return value is LB_ERR if an error occurs.

Example This example associates a handle of a 64-byte memory object with each item in a list box:

```
HGLOBAL hLBData;
LPSTR   lpLBData;
HWND    hListBox;
WPARAM  nIndex;

case WM_INITDIALOG:
```

```
if ((hLBData = GlobalAlloc(GMEM_MOVEABLE, 64))) {
    if ((lpLBData = GlobalLock(hLBData))) {
        .
        . /* Store the data in the memory object. */
        .
        GlobalUnlock(hLBData);
    }
}
SendMessage(hListBox, LB_SETITEMDATA, nIndex,
    MAKELONG(hLBData, 0));
```

See Also LB_ADDSTRING, LB_GETITEMDATA, LB_INSERTSTRING

LB_SETITEMHEIGHT

3.1

```
LB_SETITEMHEIGHT
wParam = (WPARAM) index;        /* item index */
lParam = MAKELPARAM(cyItem, 0); /* item height */
```

An application sends an LB_SETITEMHEIGHT message to set the height of items in a list box. If the list box has the LBS_OWNERDRAWVARIABLE style, this message sets the height of the item specified by the *wParam* parameter. Otherwise, this message sets the height of all items in the list box.

Parameters

index
Value of *wParam*. Specifies the zero-based index of the item in the list box. This parameter is used only if the list box has the LBS_OWNERDRAWVARIABLE style; otherwise, it should be set to zero.

cyItem
Value of the low-order word of *lParam*. Specifies the height, in pixels, of the item.

Return Value The return value is LB_ERR if the index or height is invalid.

Example This example sends an LB_SETITEMHEIGHT message to set the height of the items in a list box:

```
LPARAM lpmHeight;

SendDlgItemMessage(hdlg, ID_MYLISTBOX, LB_SETITEMHEIGHT,
    0, lpmHeight);
```

See Also LB_GETITEMHEIGHT

LB_SETSEL

2.x

```
LB_SETSEL
wParam = (WPARAM) (BOOL) fSelect;   /* selection flag */
lParam = MAKELPARAM(index, 0);      /* item index     */
```

An application sends an LB_SETSEL message to select a string in a multiple-selection list box.

Parameters

fSelect
Value of *wParam*. Specifies how to set the selection. If the *fSelect* parameter is TRUE, the string is selected and highlighted; if *fSelect* is FALSE, the highlight is removed and the string is no longer selected.

index
Value of the low-order word of *lParam*. Specifies the zero-based index of the string to set. If the *index* parameter is −1, the selection is added to or removed from all strings, depending on the value of *fSelect*.

Return Value

The return value is LB_ERR if an error occurs.

Comments

This message should be used only with multiple-selection list boxes.

See Also

LB_GETSEL

LB_SETTABSTOPS

3.0

```
LB_SETTABSTOPS
wParam = (WPARAM) cTabs;              /* number of tab stops       */
lParam = (LPARAM) (int FAR*) lpTabs; /* address of tab-stop array */
```

An application sends an LB_SETTABSTOPS message to set the tab-stop positions in a list box.

Parameters

cTabs
Value of *wParam*. Specifies the number of tab stops in the list box.

lpTabs
Value of *lParam*. Points to the first member of an array of integers containing the tab stops, in dialog box units. The tab stops must be sorted in increasing order; back tabs are not allowed.

Return Value

The return value is nonzero if all the tabs were set; otherwise, the return value is zero.

Comments

To respond to the LB_SETTABSTOPS message, the list box must have been created with the LBS_USETABSTOPS style.

If the *cTabs* parameter is zero and the *lpTabs* parameter is NULL, the default tab stop is two dialog box units.

If *cTabs* is 1, the edit control will have tab stops separated by the distance specified by *lpTabs*.

If *lpTabs* points to more than a single value, a tab stop will be set for each value in *lpTabs*, up to the number specified by *cTabs*.

A dialog box unit is a horizontal or vertical distance. One horizontal dialog box unit is equal to one-fourth of the current dialog box base width unit. The dialog box base units are computed based on the height and width of the current system font. The **GetDialogBaseUnits** function returns the current dialog box base units, in pixels.

LB_SETTOPINDEX

3.0

```
LB_SETTOPINDEX
wParam = (WPARAM) index;        /* item index             */
lParam = 0L;                    /* not used, must be zero */
```

An application sends an LB_SETTOPINDEX message to ensure that a particular item in a list box is visible.

Parameters

index
Value of *wParam*. Specifies the zero-based index of the item in the list box.

Return Value

The return value is LB_ERR if an error occurs.

Comments

The system scrolls the list box so that either the specified item appears at the top of the list box or the maximum scroll range has been reached.

Example

This example searches for an item in a list box that matches the string "my string" and, if a match is found, ensures that the item is visible:

```
int iIndex;


iIndex = (int) SendMessage(hMyListbox, LB_FINDSTRING, -1,
    (LPARAM) (LPCSTR) "my string");

if (iIndex != LB_ERR)
  SendMessage(hMyListbox, LB_SETTOPINDEX, (WPARAM) iIndex, 0L);
```

See Also

LB_GETTOPINDEX

STM_GETICON

3.1

```
STM_GETICON
wParam = 0;      /* not used, must be zero */
lParam = 0L;     /* not used, must be zero */
```

An application sends an STM_GETICON message to retrieve the handle of the icon associated with an icon resource.

Parameters

This message has no parameters.

Return Value

The return value is the icon handle if the operation is successful, or it is zero if the icon has no associated icon resource or if an error occurred.

Example

This example gets the handle of the icon associated with an icon resource:

```
HICON hIcon;

hIcon = (HICON) SendDlgItemMessage(hdlg, IDD_ICON,
    STM_GETICON, 0, 0L);
```

See Also

STM_SETICON

STM_SETICON

3.1

```
STM_SETICON
wParam = (WPARAM) (HICON) hicon;     /* handle of the icon     */
lParam = 0L;                         /* not used, must be zero */
```

An application sends an STM_SETICON message to associate an icon with an icon resource.

Parameters

hicon
Value of *wParam*. Identifies the icon to associate with the icon resource.

Return Value

The return value is the handle of the icon that was previously associated with the icon resource, or it is zero if an error occurred.

Example

This example associates the system-defined question-mark icon with an icon resource:

```
HICON hIcon, hOldIcon;

hIcon = LoadIcon((HANDLE) NULL, IDI_QUESTION);
hOldIcon = (HICON) SendDlgItemMessage(hdlg, IDD_ICON,
    STM_SETICON, hIcon, 0L);
```

See Also

STM_GETICON

WM_ACTIVATE

2.x

```
WM_ACTIVATE
fActive = wParam;                    /* activation flag */
fMinimized = (BOOL) HIWORD(lParam); /* minimized flag  */
hwnd = (HWND) LOWORD(lParam);        /* window handle   */
```

The WM_ACTIVATE message is sent when a window is being activated or deactivated. This message is sent first to the window procedure of the main window being deactivated and then to the window procedure of the main window being activated.

Parameters

fActive
Value of *wParam*. Specifies whether the window is being activated or deactivated. It can be one of the following values:

Value	Description
WA_INACTIVE	The window is being deactivated.
WA_ACTIVE	The window is being activated through some method other than a mouse click (for example, by use of the keyboard interface to select the window).
WA_CLICKACTIVE	The window is being activated by a mouse click.

fMinimized
Value of the high-order word of *lParam*. Specifies the minimized state of the window being activated or deactivated. A nonzero value indicates the window is minimized.

hwnd
Value of the low-order word of *lParam*. Identifies the window being activated or deactivated. This handle can be NULL.

Return Value

An application should return zero if it processes this message.

Comments

If the window is activated with a mouse click, it also receives a WM_MOUSEACTIVATE message.

Example

This example sets the input focus while processing the WM_ACTIVATE message:

```
case WM_ACTIVATE:

    if (wParam && !HIWORD(lParam))
        SetFocus(hwnd);
    break;
```

See Also

WM_MOUSEACTIVATE, WM_NCACTIVATE

WM_ACTIVATEAPP

2.x

```
WM_ACTIVATEAPP
fActive = (BOOL) wParam;        /* the activation/deactivation flag */
htask = (HTASK) LOWORD(lParam); /* task handle                      */
```

The WM_ACTIVATEAPP message is sent when a window is about to be activated and that window belongs to a different task than the active window. The message is sent to all top-level windows of the task being activated and to all top-level windows of the task being deactivated.

Parameters

fActive
Value of *wParam*. Specifies whether the window is being activated or deactivated. A nonzero value means the window is being activated. A zero value means the window is being deactivated.

htask
Value of the low-order word of *lParam*. Specifies a task handle. If the *fActive* parameter is nonzero, the handle identifies the task that owns the window being deactivated. If *fActive* is zero, the handle identifies the task that owns the window being activated.

Return Value

An application should return zero if it processes this message.

See Also

WM_ACTIVATE

WM_ASKCBFORMATNAME

2.x

```
WM_ASKCBFORMATNAME
wParam = (WPARAM) cbMax;            /* maximum bytes to copy  */
lParam = (LPARAM) lpszFormatName;   /* address of format name */
```

A clipboard viewer application sends a WM_ASKCBFORMATNAME message to the clipboard owner when the clipboard contains the data handle of the CF_OWNERDISPLAY format (that is, when the clipboard owner should display the clipboard contents).

Parameters

cbMax
Value of *wParam*. Specifies the maximum number of bytes to copy.

lpszFormatName
Value of *lParam*. Points to the buffer where the copy of the format name is to be stored.

Return Value

An application should return zero if it processes this message.

Comments

The clipboard owner should copy the name of the CF_OWNERDISPLAY format into the specified buffer, not exceeding the maximum number of bytes.

See Also

WM_PAINTCLIPBOARD

WM_CANCELMODE

2.x

```
WM_CANCELMODE
```

The WM_CANCELMODE message is sent to inform a window to cancel any internal mode. This message is sent to the focus window when a dialog box or message box is displayed, giving the focus window the opportunity to cancel modes such as mouse capture.

Parameters

This message has no parameters.

Return Value

An application should return zero if it processes this message.

Comments

The **DefWindowProc** function processes this message by calling the **ReleaseCapture** function. **DefWindowProc** does not cancel any other modes.

See Also

DefWindowProc, **ReleaseCapture**

WM_CHANGECBCHAIN

2.x

```
WM_CHANGECBCHAIN
hwndRemoved = (HWND) wParam;        /* handle of removed window */
hwndNext = (HWND) LOWORD(lParam);   /* handle of next window    */
```

The WM_CHANGECBCHAIN message notifies the first window in the clipboard-viewer chain that a window is being removed from the chain.

Parameters

hwndRemoved
Value of *wParam*. Identifies the window that is being removed from the clipboard-viewer chain.

hwndNext
Value of the low-order word of *lParam*. Identifies the window that follows the window being removed from the clipboard-viewer chain.

Return Value

An application should return zero if it processes this message.

Comments

Each window that receives the WM_CHANGECBCHAIN message should call the **SendMessage** function to pass the message on 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 *hwndNext* parameter becomes the next window and clipboard messages are passed on to it.

See Also

ChangeClipboardChain, SendMessage

WM_CHAR

2.x

```
WM_CHAR
nVKey = wParam;              /* virtual-key code */
dwKeyData = (DWORD) lParam;  /* key data         */
```

The WM_CHAR message is sent when a WM_KEYUP message and a WM_KEYDOWN message are translated. The WM_CHAR message contains the value of the key being pressed or released.

Parameters

nVKey

Value of *wParam*. Specifies the virtual-key code value of the key.

dwKeyData

Value of *lParam*. Specifies the repeat count, scan code, extended key, context code, previous key state, and key-transition state, as shown in the following table:

Bit	Description
0–15	Specifies the repeat count. The value is the number of times the keystroke is repeated as a result of the user holding down the key.
16–23	Specifies the scan code. The value depends on the original equipment manufacturer (OEM).
24	Specifies whether the key is an extended key, such as a function key or a key on the numeric keypad. The value is 1 if it is an extended key; otherwise, it is 0.
25–26	Not used.
27–28	Used internally by Windows.
29	Specifies the context code. The value is 1 if the ALT key is held down while the key is pressed; otherwise, the value is 0.
30	Specifies the previous key state. The value is 1 if the key is down before the message is sent, or it is 0 if the key is up.
31	Specifies the key-transition state. The value is 1 if the key is being released, or it is 0 if the key is being pressed.

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

Comments Because there is not necessarily a one-to-one correspondence between keys pressed and character messages generated, the information in the high-order word of the *dwKeyData* parameter is usually not useful to applications. The information in the high-order word applies only to the most recent WM_KEYUP or WM_KEYDOWN message that precedes the posting of the character message.

For IBM Enhanced 101- and 102-key keyboards, enhanced keys are the right ALT key and the right CTRL key on the main section of the keyboard; the INS, DEL, HOME, END, PAGE UP, PAGE DOWN, and arrow keys in the clusters to the left of the numeric keypad; and the division (/) and ENTER keys on the numeric keypad. Some other keyboards may support the extended-key bit in the *dwKeyData* parameter.

See Also WM_KEYDOWN, WM_KEYUP

WM_CHARTOITEM

3.0

```
WM_CHARTOITEM
nKey = wParam;                          /* key value       */
hwndListBox = (HWND) LOWORD(lParam);    /* list box handle */
iCaretPos = HIWORD(lParam);             /* caret position  */
```

The WM_CHARTOITEM message is sent by a list box with the LBS_WANTKEYBOARDINPUT style to its owner in response to a WM_CHAR message.

Parameters

nKey
Value of *wParam*. Specifies the value of the key the user pressed.

hwndListBox
Value of the low-order word of *lParam*. Identifies the list box.

iCaretPos
Value of the high-order word of *lParam*. Specifies the current caret position.

Return Value The return value specifies the action that the application performed in response to the message. A return value of –2 indicates that the application handled all aspects of selecting the item and requires no further action by the list box. A return value of –1 indicates that the list box should perform the default action in response to the keystroke. A return value of 0 or greater specifies the zero-based index of an item in the list box and indicates that the list box should perform the default action for the keystroke on the given item.

Comments Only owner-drawn list boxes that do not have the LBS_HASSTRINGS style can receive this message.

See Also WM_CHAR, WM_VKEYTOITEM

WM_CHILDACTIVATE

2.x

```
WM_CHILDACTIVATE
```

The WM_CHILDACTIVATE message is sent to a multiple document interface (MDI) child window when the user clicks the window's title bar or when the window is activated, moved, or sized.

Parameters This message has no parameters.

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

See Also **MoveWindow**, **SetWindowPos**

WM_CHOOSEFONT_GETLOGFONT

3.1

```
WM_CHOOSEFONT_GETLOGFONT
wParam = 0;                    /* not used, must be zero         */
lplf = (LPLOGFONT) lParam;     /* address of a LOGFONT structure */
```

An application sends a WM_CHOOSEFONT_GETLOGFONT message to the Font dialog box created by the **ChooseFont** function to retrieve the current **LOGFONT** structure.

Parameters *lplf*

Points to a **LOGFONT** structure that receives information about the current logical font.

Return Value This message does not return a value.

Comments An application uses this message to retrieve the **LOGFONT** structure while the Font dialog box is open. When the user closes the dialog box, the **ChooseFont** function receives information about the **LOGFONT** structure.

See Also WM_GETFONT

WM_CLEAR

2.x

```
WM_CLEAR
wParam = 0;      /* not used, must be zero */
lParam = 0L;     /* not used, must be zero */
```

An application sends a WM_CLEAR message to an edit control or combo box to delete (clear) the current selection, if any, in the edit control.

Parameters This message has no parameters.

Return Value The return value is nonzero if this message is sent to an edit control or a combo box.

Comments The deletion performed by the WM_CLEAR message can be undone by sending the edit control an EM_UNDO message.

To delete the current selection and place the deleted contents into the clipboard, use the WM_CUT message.

Example This example sends an EM_SETSEL message to select the entire contents of an edit control. It then sends a WM_CLEAR message to delete the contents of the edit control.

```
SendDlgItemMessage(hdlg, ID_MYEDITCONTROL,
    EM_SETSEL, 0, MAKELONG(0, -1));
SendDlgItemMessage(hdlg, ID_MYEDITCONTROL,
    WM_CLEAR, 0, 0L);
```

See Also EM_UNDO, WM_COPY, WM_CUT, WM_PASTE

WM_CLOSE

2.x

```
WM_CLOSE
wParam = 0;      /* not used, must be zero */
lParam = 0L;     /* not used, must be zero */
```

The WM_CLOSE message is sent as a signal that a window or an application should terminate. An application can prompt the user for confirmation prior to destroying the window by processing the WM_CLOSE message and calling the **DestroyWindow** function only if the user confirms the choice.

Parameters

This message has no parameters.

Return Value

An application should return zero if it processes this message.

Example

This example processes a WM_CLOSE message and requests confirmation from the user before terminating the application:

```
case WM_CLOSE:
    if (MessageBox(hwnd, "Are you sure you want to exit?", "MyApp",
            MB_ICONQUESTION | MB_OKCANCEL) == IDOK)
        DestroyWindow(hwnd);
    return 0L;
```

See Also

DestroyWindow, **PostQuitMessage** WM_DESTROY, WM_QUIT

WM_COMMAND

2.x

```
WM_COMMAND
idItem = wParam;        /* control or menu item identifier      */
hwndCtl = (HWND) LOWORD(lParam); /* handle of control           */
wNotifyCode = HIWORD(lParam);    /* notification message        */
```

The WM_COMMAND message is sent to a window when the user selects an item from a menu, when a control sends a notification message to its parent window, or when an accelerator keystroke is translated.

Parameters

idItem
Value of *wParam*. Specifies the identifier of the menu item or control.

hwndCtl
Value of the low-order word of *lParam*. Identifies the control sending the message if the message is from a control. Otherwise, this parameter is zero.

wNotifyCode
Value of the high-order word of *lParam*. Specifies the notification message if the message is from a control. If the message is from an accelerator, this parameter is 1. If the message is from a menu, this parameter is 0.

Return Value

An application should return zero if it processes this message.

Comments

Accelerator keystrokes that are defined to select items from the System menu (sometimes referred to as the Control menu) are translated into WM_SYSCOMMAND messages.

If an accelerator keystroke that corresponds to a menu item occurs when the window that owns the menu is minimized, no WM_COMMAND message is sent. However, if an accelerator 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 minimized.

Example

This example creates an Options dialog box in response to a WM_COMMAND message sent as a result of a menu selection:

```
FARPROC lpProc;

case WM_COMMAND:
    switch (wParam) {
        case IDM_OPTIONS:
            lpProc = MakeProcInstance(OptionsProc, hInstance);
            DialogBox(hInstance, "OptionsBox", hwnd, lpProc);
            FreeProcInstance(lpProc);
            break;
        .
        . /* Process other menu commands. */
        .
    }
    break;
```

See Also

WM_SYSCOMMAND

WM_COMMNOTIFY

3.1

```
WM_COMMNOTIFY
idDevice = wParam;              /* communication-device ID  */
nNotifyStatus = LOWORD(lParam); /* notification-status flag */
```

The WM_COMMNOTIFY message is posted by a communication device driver whenever a COM port event occurs. The message indicates the status of a window's input or output queue.

Parameters

idDevice

Value of *wParam*. Specifies the identifier of the communication device that is posting the notification message.

nNotifyStatus

Value of the low-order word of *lParam*. Specifies the notification status in the low-order word. The notification status may be one or more of the following flags:

Value	Meaning
CN_EVENT	Indicates that an event has occurred that was enabled in the event word of the communication device. This event was enabled by a call to the **SetCommEventMask** function. The application should call the **GetCommEventMask** function to determine which event occurred and to clear the event.
CN_RECEIVE	Indicates that at least *cbWriteNotify* bytes are in the input queue. The *cbWriteNotify* parameter is a parameter of the **EnableCommNotification** function.
CN_TRANSMIT	Indicates that fewer than *cbOutQueue* bytes are in the output queue waiting to be transmitted. The *cbOutQueue* parameter is a parameter of the **EnableCommNotification** function.

Return Value

An application should return zero if it processes this message.

Comments

This message is sent only when the event word changes for the communication device. The application that sends WM_COMMNOTIFY must clear each event to be sure of receiving future notifications.

See Also

EnableCommNotification

WM_COMPACTING

3.0

```
WM_COMPACTING
wCompactRatio = wParam;  /* compacting ratio */
```

The WM_COMPACTING message is sent to all top-level windows when Windows detects that more than 12.5 percent of system time over a 30- to 60-second interval is being spent compacting memory. This indicates that system memory is low.

Parameters

wCompactRatio
Value of *wParam*. Specifies the ratio of central processing unit (CPU) time currently spent by Windows compacting memory to CPU time currently spent by Windows performing other operations. For example, 0x8000 represents 50 percent of CPU time spent compacting memory.

Return Value

An application should return zero if it processes this message.

Comments

When an application receives this message, it should free as much memory as possible, taking into account the current level of activity of the application and the total number of applications running with Windows. The application can call the **GetNumTasks** function to determine how many applications are running.

See Also

GetNumTasks

WM_COMPAREITEM

3.0

```
WM_COMPAREITEM
idCtl = wParam;                                /* control identifier */
lpcis = (const COMPAREITEMSTRUCT FAR*) lParam; /* structure          */
```

The WM_COMPAREITEM message determines the relative position of a new item in the sorted list of an owner-drawn combo box or list box. Whenever the application adds a new item, Windows sends this message to the owner of a combo box or list box created with the CBS_SORT or LBS_SORT style.

Parameters

idCtl

Value of *wParam*. Specifies the identifier of the control that sent the WM_COMPAREITEM message.

lpcis

Value of *lParam*. Points to a **COMPAREITEMSTRUCT** data structure that contains the identifiers and application-supplied data for two items in the combo box or list box. The **COMPAREITEMSTRUCT** structure has the following form:

```
typedef struct tagCOMPAREITEMSTRUCT {    /* cis */
    UINT  CtlType;
    UINT  CtlID;
    HWND  hwndItem;
    UINT  itemID1;
    DWORD itemData1;
    UINT  itemID2;
    DWORD itemData2;
} COMPAREITEMSTRUCT;
```

Return Value

The return value indicates the relative position of the two items. It may be any of the following values:

Value	Meaning
–1	Item 1 precedes item 2 in the sorted order.
0	Item 1 and item 2 are equivalent in the sorted order.
1	Item 1 follows item 2 in the sorted order.

Comments

When the owner of an owner-drawn combo box or list box receives this message, the owner returns a value indicating which of the items specified in the **COMPAREITEMSTRUCT** structure should appear before the other. Typically, Windows sends this message several times until it determines the exact position for the new item.

See Also

COMPAREITEMSTRUCT

WM_COPY

2.x

```
WM_COPY
wParam = 0;      /* not used, must be zero */
lParam = 0L;     /* not used, must be zero */
```

An application sends a WM_COPY message to an edit control or combo box to copy the current selection to the clipboard in CF_TEXT format.

Parameters

This message has no parameters.

Return Value

The return value is nonzero if this message is sent to an edit control or a combo box.

Example

This example sends an EM_SETSEL message to select the entire contents of an edit control. It then sends a WM_COPY message to copy the contents of the edit control to the clipboard.

```
SendDlgItemMessage(hdlg, ID_MYEDITCONTROL,
    EM_SETSEL, 0, MAKELONG(0, -1));
SendDlgItemMessage(hdlg, ID_MYEDITCONTROL,
    WM_COPY, 0, 0L);
```

See Also

WM_CLEAR, WM_CUT, WM_PASTE

WM_CREATE

2.x

```
WM_CREATE
lpcs = (CREATESTRUCT FAR*) lParam;      /* structure address          */
```

The WM_CREATE message is sent when an application requests that a window be created by calling the **CreateWindowEx** or **CreateWindow** function. The window procedure for the new window receives this message after the window is created but before the window becomes visible. The message is sent to the window before the **CreateWindowEx** or **CreateWindow** function returns.

Parameters

lpcs

Value of *lParam*. Points to a **CREATESTRUCT** data structure containing information about the window being created. The members of the **CREATESTRUCT** structure are identical to the parameters of the **CreateWindowEx** function.

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

```
typedef struct tagCREATESTRUCT {    /* cs */
    void FAR* lpCreateParams;
    HINSTANCE hInstance;
    HMENU     hMenu;
    HWND      hwndParent;
    int       cy;
    int       cx;
    int       y;
    int       x;
    LONG      style;
    LPCSTR    lpszName;
    LPCSTR    lpszClass;
    DWORD     dwExStyle;
} CREATESTRUCT;
```

Return Value

If an application processes this message, it should return 0 to continue creation of the window. If the application returns –1, the window will be destroyed and the **CreateWindowEx** or **CreateWindow** function will return a NULL handle.

See Also

CreateWindow, **CreateWindowEx**, WM_NCCREATE

WM_CTLCOLOR

2.x

```
WM_CTLCOLOR
hdcChild = (HDC) wParam;            /* child-window display context */
hwndChild = (HWND) LOWORD(lParam);  /* handle of child window       */
nCtlType = (int) HIWORD(lParam);    /* type of control              */
```

The WM_CTLCOLOR message is sent to the parent of a system-defined control class or a message box when the control or message box is about to be drawn. The following controls send this message:

Combo boxes
Edit controls
List boxes
Buttons
Static controls
Scroll bars

Parameters

hdcChild
Value of *wParam.* Identifies the display context for the child window.

hwndChild
Value of the low-order word of *lParam.* Identifies the child window.

nCtlType
Value of the high-order word of *lParam.* Specifies the type of the control. This parameter can be one of the following values:

Value	Meaning
CTLCOLOR_BTN	Button
CTLCOLOR_DLG	Dialog box
CTLCOLOR_EDIT	Edit control
CTLCOLOR_LISTBOX	List box
CTLCOLOR_MSGBOX	Message box
CTLCOLOR_SCROLLBAR	Scroll bar
CTLCOLOR_STATIC	Static control

Return Value

If an application processes the WM_CTLCOLOR message, it must return a handle to the brush that is to be used for painting the control background or it must return NULL.

Comments

To change the text color, the application should call the **SetTextColor** function with the desired red, green, and blue (RGB) values.

To change the background color of a single-line edit control, the application must set the brush handle in both the CTLCOLOR_EDIT and CTLCOLOR_MSGBOX message codes, and the application must call the **SetBkColor** function in response to the CTLCOLOR_EDIT code.

The return value from this message has no effect on a button with the BS_PUSHBUTTON or BS_DEFPUSHBUTTON style.

Example

This example creates a green brush and passes the handle of the brush to a single-line edit control in response to a WM_CTLCOLOR message:

```
static HBRUSH hbrGreen;

switch(msg) {
    case WM_INITDIALOG:

        /* Create a green brush */

        hbrGreen = CreateSolidBrush(RGB(0, 255, 0));
        return TRUE;
```

```
        case WM_CTLCOLOR:
            switch(HIWORD(lParam)) {
                case CTLCOLOR_EDIT:

                    /* Set text to white and background to green */

                    SetTextColor((HDC) wParam, RGB(255, 255, 255));
                    SetBkColor((HDC) wParam, RGB(0, 255, 0));
                    return hbrGreen;
                    break;

                case CTLCOLOR_MSGBOX:

                    /*
                     * For single-line edit controls, this code must be
                     * processed so that the background color of the format
                     * rectangle will also be painted with the new color.
                     */

                    return hbrGreen;
            }
            return (HBRUSH) NULL;
}
```

See Also **SetBkColor**

WM_CUT

2.x

```
WM_CUT
wParam = 0;     /* not used, must be zero */
lParam = 0L;    /* not used, must be zero */
```

An application sends a WM_CUT message to an edit control or combo box to delete (cut) the current selection, if any, in the edit control and copy the deleted text to the clipboard in CF_TEXT format.

Parameters This message has no parameters.

Return Value The return value is nonzero if this message is sent to an edit control or a combo box.

Comments An EM_UNDO message can be sent to the edit control to undo the deletion performed by the WM_CUT message.

To delete the current selection without placing the deleted text onto the clipboard, use the WM_CLEAR message.

Example

This example sends an EM_SETSEL message to select the entire contents of an edit control. It then sends a WM_CUT message to delete the contents of the edit control and to copy the deleted text to the clipboard.

```
SendDlgItemMessage(hdlg, ID_MYEDITCONTROL,
    EM_SETSEL, 0, MAKELONG(0, -1));
SendDlgItemMessage(hdlg, ID_MYEDITCONTROL,
    WM_CUT, 0, 0L);
```

See Also

WM_CLEAR, WM_COPY, WM_PASTE

WM_DDE_ACK

2.x

```
#include <dde.h>

WM_DDE_ACK
wParam = (WPARAM) hwnd;            /* handle of posting window      */
lParam = MAKELPARAM(wLow, wHigh);  /* depending on received message */
```

The WM_DDE_ACK message notifies an application of the receipt and processing of a WM_DDE_INITIATE, WM_DDE_EXECUTE, WM_DDE_DATA, WM_DDE_ADVISE, WM_DDE_UNADVISE, or WM_DDE_POKE message, and in some cases, of a WM_DDE_REQUEST message.

Parameters

hwnd

Value of *wParam.* Specifies the handle of the window posting the message.

wLow

Value of the low-order word of *lParam.* Specifies data as follows, depending on the message to which the WM_DDE_ACK message is responding:

Message	Parameter	Description
WM_DDE_INITIATE	*aApplication*	An atom that contains the name of the replying application.
WM_DDE_EXECUTE and all other messages	*wStatus*	A series of flags that indicate the status of the response.

wHigh
Value of high-order word of *lParam*. Specifies data as follows, depending on the message to which the WM_DDE_ACK message is responding:

Message	Parameter	Description
WM_DDE_INITIATE	*aTopic*	An atom that contains the topic with which the replying server window is associated.
WM_DDE_EXECUTE	*hCommands*	A handle that identifies the data item containing the command string.
All other messages	*aItem*	An atom that specifies the data item for which the response is sent.

Return Value

This message does not return a value.

Comments

The *wStatus* word consists of a **DDEACK** data structure. The **DDEACK** structure has the following form:

```
#include <dde.h>

typedef struct tagDDEACK {  /* ddeack */
    WORD bAppReturnCode:8,
         reserved:6,
         fBusy:1,
         fAck:1;
} DDEACK;
```

For a full description of this structure, see Chapter 3, "Structures."

Posting

Except in response to the WM_DDE_INITIATE message, the application posts the WM_DDE_ACK message by calling the **PostMessage** function, not the **SendMessage** function. When responding to WM_DDE_INITIATE, the application sends the WM_DDE_ACK message by calling **SendMessage**.

When acknowledging any message with an accompanying *aItem* atom, the application posting WM_DDE_ACK can either reuse the *aItem* atom that accompanied the original message or delete it and create a new one.

When acknowledging WM_DDE_EXECUTE, the application that posts WM_DDE_ACK should reuse the *hCommands* object that accompanied the original WM_DDE_EXECUTE message.

If an application has initiated the termination of a conversation by posting WM_DDE_TERMINATE and is awaiting confirmation, the waiting application should not acknowledge (positively or negatively) any subsequent messages sent by the other application. The waiting application should delete any atoms or shared memory objects received in these intervening messages (but should not delete the atoms in response to the WM_DDE_ACK message).

Receiving

The application that receives WM_DDE_ACK should delete all atoms accompanying the message.

If the application receives WM_DDE_ACK in response to a message with an accompanying *hData* object, the application should delete the *hData* object.

If the application receives a negative WM_DDE_ACK message posted in reply to a WM_DDE_ADVISE message, the application should delete the *hOptions* object posted with the original WM_DDE_ADVISE message.

If the application receives a negative WM_DDE_ACK message posted in reply to a WM_DDE_EXECUTE message, the application should delete the *hCommands* object posted with the original WM_DDE_EXECUTE message.

See Also **DDEACK**, **PostMessage**, WM_DDE_ADVISE, WM_DDE_DATA, WM_DDE_EXECUTE, WM_DDE_INITIATE, WM_DDE_POKE, WM_DDE_REQUEST, WM_DDE_TERMINATE, WM_DDE_UNADVISE

WM_DDE_ADVISE

2.x

```
#include <dde.h>

WM_DDE_ADVISE
wParam = (WPARAM) hwnd;                 /* handle of posting window   */
lParam = MAKELPARAM(hOptions, aItem); /* send options and data item */
```

A dynamic data exchange (DDE) client application posts the WM_DDE_ADVISE message to a DDE server application to request the server to supply an update for a data item whenever it changes.

Parameters

hwnd
Value of *wParam*. Identifies the sending window.

hOptions
Value of the low-order word of *lParam*. Specifies a handle of a global memory object that specifies how the data is to be sent.

aItem
Value of the high-order word of *lParam*. Specifies the data item being requested.

Return Value

This message does not return a value.

Comments

The global memory object identified by the *hOptions* parameter consists of a **DDEADVISE** data structure. The **DDEADVISE** data structure has the following form:

```
#include <dde.h>

typedef struct tagDDEADVISE {  /* ddeadv */
    WORD    reserved:14,
            fDeferUpd:1,
            fAckReq:1;
    short   cfFormat;
} DDEADVISE;
```

For a full description of this structure, see Chapter 3, "Structures."

If an application supports more than one clipboard format for a single topic and item, it can post multiple WM_DDE_ADVISE messages for the topic and item, specifying a different clipboard format with each message.

Posting

The application posts the WM_DDE_ADVISE message by calling the **PostMessage** function, not the **SendMessage** function.

The application allocates *hOptions* by calling the **GlobalAlloc** function with the GMEM_DDESHARE option.

The application allocates *aItem* by calling the **GlobalAddAtom** function.

If the receiving (server) application responds with a negative WM_DDE_ACK message, the posting (client) application must delete the *hOptions* object.

Receiving

The application posts the WM_DDE_ACK message to respond positively or negatively. When posting WM_DDE_ACK, the application can reuse the *aItem* atom or delete it and create a new one. If the WM_DDE_ACK message is positive, the application should delete the *hOptions* object; otherwise, the application should not delete the object.

See Also **DDEADVISE**, **GlobalAddAtom**, **GlobalAlloc**, **PostMessage**, WM_DDE_DATA, WM_DDE_REQUEST

WM_DDE_DATA

2.x

```
#include <dde.h>

WM_DDE_DATA
wParam = (WPARAM) hwnd;              /* handle of posting window    */
lParam = MAKELPARAM(hData, aItem);   /* memory object and data item */
```

A dynamic data exchange (DDE) server application posts a WM_DDE_DATA message to a DDE client application to pass a data item to the client or to notify the client of the availability of a data item.

Parameters

hwnd
Value of *wParam*. Specifies the handle of the window posting the message.

hData
Value of the low-order word of *lParam*. Identifies the global memory object containing the data and additional information. The handle should be set to NULL if the server is notifying the client that the data item value has changed during a warm link. A warm link is established when the client sends a WM_DDE_ADVISE message with the *fDeferUpd* bit set.

aItem
Value of the high-order word of *lParam*. Specifies the data item for which data or notification is sent.

Return Value This message does not return a value.

Comments The global memory object identified by the *hData* parameter consists of a **DDEDATA** structure. The **DDEDATA** structure has the following form:

```
#include <dde.h>

typedef struct tagDDEDATA {    /* ddedat */
    WORD    unused:12,
            fResponse:1,
            fRelease:1,
            reserved:1,
            fAckReq:1;
    short   cfFormat;
    BYTE    Value[1];
} DDEDATA;
```

For a full description of this structure, see Chapter 3, "Structures."

Posting

The application posts the WM_DDE_DATA message by calling the **PostMessage** function, not the **SendMessage** function.

The application allocates *hData* by calling the **GlobalAlloc** function with the GMEM_DDESHARE option.

The application allocates *aItem* by calling the **GlobalAddAtom** function.

If the receiving (client) application responds with a negative WM_DDE_ACK message, the posting (server) application must delete the *hData* object.

If the posting (server) application sets the **fRelease** member of the **DDEDATA** structure to FALSE, the posting application is responsible for deleting *hData* upon receipt of either a positive or negative acknowledgment.

The application should not set both the **fAckReq** and **fRelease** members of the **DDEDATA** structure to FALSE. If both members are set to FALSE, it is difficult for the posting (server) application to determine when to delete *hData*.

Receiving

If **fAckReq** is TRUE, the application posts the WM_DDE_ACK message to respond positively or negatively. When posting WM_DDE_ACK, the application can reuse the *aItem* atom or delete it and create a new one.

If **fAckReq** is FALSE, the application deletes the *aItem* atom.

If the posting (server) application specified *hData* as NULL, the receiving (client) application can request the server to send the actual data by posting a WM_DDE_REQUEST message.

After processing a WM_DDE_DATA message in which *hData* is not NULL, the application should delete *hData* unless either of the following conditions is true:

- The **fRelease** member is FALSE.
- The **fRelease** member is TRUE, but the receiving (client) application responds with a negative WM_DDE_ACK message.

See Also **DDEDATA**, **GlobalAddAtom**, **GlobalAlloc**, **PostMessage**, WM_DDE_ACK, WM_DDE_ADVISE, WM_DDE_POKE, WM_DDE_REQUEST

WM_DDE_EXECUTE

2.x

```
#include <dde.h>

WM_DDE_EXECUTE
wParam = (WPARAM) hwnd;                    /* handle of posting window */
lParam = MAKELPARAM(reserved, hCommands); /* commands to execute      */
```

A dynamic data exchange (DDE) client application posts a WM_DDE_EXECUTE message to a DDE server application to send a string to the server to be processed as a series of commands. The server application is expected to post a WM_DDE_ACK message in response.

Parameters

hwnd
Value of *wParam*. Identifies the sending window.

reserved
Value of the low-order word of *lParam*. Reserved; must be zero.

hCommands
Value of the high-order word of *lParam*. Identifies a global memory object containing the command(s) to be executed.

Return Value This message does not return a value.

Comments The command string is a null-terminated string, consisting of one or more *opcode* strings enclosed in single brackets ([ ]) and separated by spaces.

Each *opcode* string has the following syntax. The *parameters* list is optional.

opcode parameters

The *opcode* is any application-defined single token. It cannot include spaces, commas, parentheses, or quotation marks.

The *parameters* list can contain any application-defined value or values. Multiple parameters are separated by commas, and the entire parameter list is enclosed in parentheses. Parameters cannot include commas or parentheses except inside a quoted string. If a bracket or parenthesis character is to appear in a quoted string, it must be doubled—for example, "((".

The following are valid command strings:

```
[connect][download(query1,results.txt)][disconnect]
[query("sales per employee for each district")]
[open("sample.xlm")][run("r1c1")]
```

Posting

The application posts the WM_DDE_EXECUTE message by calling the **PostMessage** function, not the **SendMessage** function.

The application allocates *hCommands* by calling the **GlobalAlloc** function with the GMEM_DDESHARE option.

When processing a WM_DDE_ACK message posted in reply to a WM_DDE_EXECUTE message, the application that posted the original WM_DDE_EXECUTE message must delete the *hCommands* object sent back in the WM_DDE_ACK message.

Receiving

The application posts the WM_DDE_ACK message to respond positively or negatively, reusing the *hCommands* object.

See Also **PostMessage**, WM_DDE_ACK

WM_DDE_INITIATE

2.x

```
#include <dde.h>

WM_DDE_INITIATE
wParam = (WPARAM) hwnd;                    /* sending window's handle */
lParam = MAKELPARAM(aApplication, aTopic); /* application and topic   */
```

A dynamic data exchange (DDE) client application sends a WM_DDE_INITIATE message to initiate a conversation with server applications responding to the specified application and topic names.

Upon receiving this message, all server applications with names that match the *aApplication* application and that support the *aTopic* topic are expected to acknowledge it (see the WM_DDE_ACK message).

Parameters

hwnd
Value of *wParam*. Identifies the sending window.

aApplication
Value of the low-order word of *lParam*. Specifies the name of the application with which a conversation is requested. The application name cannot contain slash marks (/) or backslashes (\). These characters are reserved for future use in network implementations. If *aApplication* is NULL, a conversation with all applications is requested.

aTopic
Value of the high-order word of *lParam*. Specifies the topic for which a conversation is requested. If the topic is NULL, a conversation for all available topics is requested.

Return Value

This message does not return a value.

Comments

If *aApplication* is NULL, any application can respond. If *aTopic* is NULL, any topic is valid. Upon receiving a WM_DDE_INITIATE request with the *aTopic* parameter set to NULL, an application is expected to send a WM_DDE_ACK message for each of the topics it supports.

Sending

The application sends the WM_DDE_INITIATE message by calling the **SendMessage** function, not the **PostMessage** function. The application broadcasts the message to all windows by setting the first parameter of **SendMessage** to –1, as shown:

```
SendMessage(-1, WM_DDE_INITIATE, hwndClient, MAKELONG(aApp, aTopic));
```

If the application has already obtained the window handle of the desired server, it can send WM_DDE_INITIATE directly to the server window by passing the server's window handle as the first parameter of **SendMessage**.

The application allocates *aApplication* and *aTopic* by calling **GlobalAddAtom**.

When **SendMessage** returns, the application deletes the *aApplication* and *aTopic* atoms.

Receiving

To complete the initiation of a conversation, the application responds with one or more WM_DDE_ACK messages, where each message is for a separate topic. When sending a WM_DDE_ACK message, the application creates new *aApplication* and *aTopic* atoms; it should not reuse the atoms sent with the WM_DDE_INITIATE message.

See Also **GlobalAddAtom**, **SendMessage**, WM_DDE_ACK

WM_DDE_POKE

2.x

```
#include <dde.h>

WM_DDE_POKE
wParam = (WPARAM) hwnd;              /* handle of posting window */
lParam = MAKELPARAM(hData, aItem);   /* data handle and item     */
```

A dynamic data exchange (DDE) client application posts a WM_DDE_POKE message to a server application. A client uses this message to request the server to accept an unsolicited data item. The server is expected to reply with a WM_DDE_ACK message indicating whether it accepted the data item.

Parameters

hwnd

Value of *wParam*. Specifies the handle of the window posting the message.

hData

Value of the low-order word of *lParam*. Identifies the data being posted. The handle identifies a global memory object that contains a **DDEPOKE** data structure. The **DDEPOKE** structure has the following form:

```
#include <dde.h>

typedef struct tagDDEPOKE {  /* ddepok */
    WORD  unused:13,
          fRelease:1,
          fReserved:2;
    short cfFormat;
    BYTE  Value[1];
} DDEPOKE;
```

For a full description of this structure, see Chapter 3, "Structures."

aItem
Value of the high-order word of *lParam.* Specifies a global atom that identifies the data item being offered to the server.

Return Value

This message does not return a value.

Comments

Posting

The posting (client) application should do the following:

- Use the **PostMessage** function to post the WM_DDE_POKE message.
- Use the **GlobalAlloc** function with the GMEM_DDESHARE option to allocate memory for the data.
- Use the **GlobalAddAtom** function to create the atom for the data item.
- Delete the global memory object if the server application responds with a negative WM_DDE_ACK message.
- Delete the global memory object if the client has set the **fRelease** member of the **DDEPOKE** structure to FALSE and the server responds with either a positive or negative WM_DDE_ACK.

Receiving

The receiving (server) application should do the following:

- Post the WM_DDE_ACK message to respond positively or negatively. When posting WM_DDE_ACK, reuse the data-item atom or delete it and create a new one.
- Delete the global memory object after processing WM_DDE_POKE unless either the **fRelease** flag was set to FALSE or the **fRelease** flag was set to TRUE but the server has responded with a negative WM_DDE_ACK message.

See Also

DDEPOKE, **GlobalAlloc**, **PostMessage**, WM_DDE_ACK, WM_DDE_DATA

WM_DDE_REQUEST

2.x

```
#include <dde.h>

WM_DDE_REQUEST
wParam = (WPARAM) hwnd;                /* handle of posting window  */
lParam = MAKELPARAM(cfFormat, aItem); /* clipboard format and item */
```

A dynamic data exchange (DDE) client application posts a WM_DDE_REQUEST message to a DDE server application to request the value of a data item.

Parameters

hwnd
Value of *wParam*. Identifies the sending window.

cfFormat
Value of the low-order word of *lParam*. Specifies a standard or registered clipboard format number.

aItem
Value of the high-order word of *lParam*. Specifies which data item is being requested from the server.

Return Value

This message does not return a value.

Comments

Posting

The application posts the WM_DDE_REQUEST message by calling the **PostMessage** function, not the **SendMessage** function.

The application allocates *aItem* by calling the **GlobalAddAtom** function.

Receiving

If the receiving (server) application can satisfy the request, it responds with a WM_DDE_DATA message containing the requested data. Otherwise, it responds with a negative WM_DDE_ACK message.

When responding with either a WM_DDE_DATA or WM_DDE_ACK message, the application can reuse the *aItem* atom or delete it and create a new one.

See Also

GlobalAddAtom, **PostMessage**, WM_DDE_ACK

WM_DDE_TERMINATE

2.x

```
#include <dde.h>

WM_DDE_TERMINATE
wParam = (WPARAM) hwnd; /* handle of posting window */
lParam = 0L;            /* not used, must be zero   */
```

A dynamic data exchange (DDE) application (client or server) posts a WM_DDE_TERMINATE message to terminate a conversation.

Parameters

hwnd
Value of *wParam.* Identifies the sending window.

Return Value

This message does not return a value.

Comments

Posting

The application posts the WM_DDE_TERMINATE message by calling the **PostMessage** function, not the **SendMessage** function.

While waiting for confirmation of the termination, the posting application should not acknowledge any other messages sent by the receiving application. If the posting application receives messages (other than WM_DDE_TERMINATE) from the receiving application, it should delete any atoms or shared memory objects accompanying the messages.

Receiving

The application responds by posting a WM_DDE_TERMINATE message.

See Also

PostMessage

WM_DDE_UNADVISE

2.x

```
#include <dde.h>

WM_DDE_UNADVISE
wParam = (WPARAM) hwnd;                  /* handle of posting window   */
lParam = MAKELPARAM(cfFormat, aItem);    /* clipboard format and item  */
```

A dynamic data exchange (DDE) client application posts a WM_DDE_UNADVISE message to inform a server application that the specified item or a particular clipboard format for the item should no longer be updated. This terminates the warm or hot link for the specified item.

Parameters

hwnd
Value of *wParam*. Identifies the sending window.

cfFormat
Value of the low-order word of *lParam*. Specifies the clipboard format of the item for which the update request is being retracted. When the *cfFormat* parameter is NULL, all active WM_DDE_ADVISE conversations for the item are to be terminated.

aItem
Value of the high-order word of *lParam*. Specifies the item for which the update request is being retracted. When *aItem* is NULL, all active WM_DDE_ADVISE conversations associated with the client are to be terminated.

Return Value

This message does not return a value.

Comments

Posting

The application posts the WM_DDE_UNADVISE message by calling the **PostMessage** function, not the **SendMessage** function.

The application allocates *aItem* by calling the **GlobalAddAtom** function.

Receiving

The application posts the WM_DDE_ACK message to respond positively or negatively. When posting WM_DDE_ACK, the application can reuse the *aItem* atom or delete it and create a new one.

See Also

GlobalAddAtom, **PostMessage**, WM_DDE_ACK

WM_DEADCHAR

2.x

```
WM_DEADCHAR
chDeadKey = wParam;          /* dead-key character */
dwKeyData = (DWORD) lParam;  /* key data           */
```

The WM_DEADCHAR message is sent when a WM_KEYUP message and a WM_KEYDOWN message are translated. It specifies the character value of a dead key. A dead key is a key, such as the umlaut (double-dot) character, that is combined with other characters to form a composite character. For example, the umlaut-O character consists of the dead key, umlaut, and the O key.

Parameters

chDeadKey
Value of *wParam*. Specifies the dead-key character value.

dwKeyData
Value of *lParam*. Specifies the repeat count, scan code, extended key, context code, previous key state, and key-transition state, as shown in the following table:

Bit	Description
0–15	Specifies the repeat count. The value is the number of times the keystroke is repeated as a result of the user holding down the key.
16–23	Specifies the scan code. The value depends on the original equipment manufacturer (OEM).
24	Specifies whether the key is an extended key, such as a function key or a key on the numeric keypad. The value is 1 if it is an extended key; otherwise, it is 0.
25–26	Not used.
27–28	Used internally by Windows.
29	Specifies the context code. The value is 1 if the ALT key is held down while the key is pressed; otherwise, the value is 0.
30	Specifies the previous key state. The value is 1 if the key is down before the message is sent, or it is 0 if the key is up.
31	Specifies the key-transition state. The value is 1 if the key is being released, or it is 0 if the key is being pressed.

Return Value

An application should return zero if it processes this message.

Comments

An application typically uses the WM_DEADCHAR message to give the user feedback about each key pressed. For example, an application can display the accent in the current character position without moving the caret.

Because there is not necessarily a one-to-one correspondence between keys pressed and character messages generated, the information in the high-order word

of the *dwKeyData* parameter is usually not useful to applications. The information in the high-order word applies only to the most recent WM_KEYUP or WM_KEYDOWN message that precedes the posting of the character message.

For IBM Enhanced 101- and 102-key keyboards, enhanced keys are the right ALT key and the right CTRL key on the main section of the keyboard; the INS, DEL, HOME, END, PAGE UP, PAGE DOWN, and arrow keys in the clusters to the left of the numeric keypad; and the division (/) and ENTER keys on the numeric keypad. Some other keyboards may support the extended-key bit in the *dwKeyData* parameter.

See Also WM_KEYDOWN

WM_DELETEITEM

3.0

```
WM_DELETEITEM
idCtl = wParam;                              /* control identifier */
lpdis = (const DELETEITEMSTRUCT FAR*) lParam; /* structure      */
```

The WM_DELETEITEM message is sent to the owner of an owner-drawn list box or combo box when the list box or combo box is destroyed or when items are removed by the LB_DELETESTRING, LB_RESETCONTENT, CB_DELETESTRING, or CB_RESETCONTENT message.

Parameters

idCtl

Value of *wParam*. Specifies the identifier of the control that sent the WM_DELETEITEM message.

lpdis

Value of *lParam*. Points to a **DELETEITEMSTRUCT** structure that contains information about the item deleted from the list box. The **DELETEITEMSTRUCT** structure has the following form:

```
typedef struct tagDELETEITEMSTRUCT {    /* deli */
    UINT  CtlType;
    UINT  CtlID;
    UINT  itemID;
    HWND  hwndItem;
    DWORD itemData;
} DELETEITEMSTRUCT;
```

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

See Also CB_DELETESTRING, CB_RESETCONTENT, LB_DELETESTRING, LB_RESETCONTENT

WM_DESTROY

2.x

```
WM_DESTROY
```

The WM_DESTROY message is sent when a window is being destroyed. It is sent to the window procedure of the window being destroyed after the window is removed from the screen.

This message is sent first to the window being destroyed and then to the child windows as they are destroyed. During the processing of the WM_DESTROY message, it can be assumed that all child windows still exist.

Parameters

This message has no parameters.

Return Value

An application should return zero if it processes this message.

Comments

If the window being destroyed is part of the clipboard-viewer chain (set by calling the **SetClipboardViewer** function), the window must remove itself from the clipboard-viewer chain by calling the **ChangeClipboardChain** function before returning from the WM_DESTROY message.

Example

This example processes the WM_DESTROY message by calling the **PostQuitMessage** function:

```
case WM_DESTROY:
    PostQuitMessage(0);
    return 0L;
```

See Also

ChangeClipboardChain, **DestroyWindow**, **PostQuitMessage**, **SetClipboardViewer**, WM_CLOSE

WM_DESTROYCLIPBOARD

2.x

```
WM_DESTROYCLIPBOARD
```

The WM_DESTROYCLIPBOARD message is sent to the clipboard owner when the clipboard is emptied by a call to the **EmptyClipboard** function.

Parameters

This message has no parameters.

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

See Also **EmptyClipboard**

WM_DEVMODECHANGE

2.x

```
WM_DEVMODECHANGE
lpszDev = (LPCSTR) lParam;  /* address of device name */
```

The WM_DEVMODECHANGE message is sent to all top-level windows when the default device-mode settings have changed.

Parameters *lpszDev*
Value of *lParam*. Points to the device name specified in the Windows initialization file, WIN.INI.

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

Comments Applications that receive this message may reinitialize their device-mode settings. Applications that use the **ExtDeviceMode** function to save and restore device settings typically do not process this message.

This message is not sent when the user changes the default printer from Control Panel. In this case, a WM_WININICHANGE message is generated.

See Also **ExtDeviceMode**, WM_WININICHANGE

WM_DRAWCLIPBOARD

2.x

```
WM_DRAWCLIPBOARD
```

The WM_DRAWCLIPBOARD message is sent to the first window in the clipboard-viewer chain when the contents of the clipboard change. Only applications that have joined the clipboard-viewer chain by calling the **SetClipboardViewer** function need to process this message.

Parameters This message has no parameters.

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

Comments Each window that receives the WM_DRAWCLIPBOARD message should call the **SendMessage** function to pass the message on to the next window in the clipboard-viewer chain. The handle of the next window is returned by the **SetClipboardViewer** function; the handle may be modified in response to a WM_CHANGECBCHAIN message.

See Also **SendMessage**, **SetClipboardViewer**, WM_CHANGECBCHAIN

WM_DRAWITEM

3.0

```
WM_DRAWITEM
idCtl = (int) wParam;                       /* control identifier */
lpdis = (const DRAWITEMSTRUCT FAR*) lParam;    /* structure     */
```

The WM_DRAWITEM message is sent to the owner of an owner-drawn button, combo box, list box, or menu when a visual aspect of the button, combo box, list box, or menu has changed.

Parameters

idCtl

Value of *wParam*. Specifies the identifier of the control that sent the WM_DRAWITEM message. This parameter is zero if the message was sent by a menu.

lpdis

Value of *lParam*. Points to a **DRAWITEMSTRUCT** structure that contains information about the item to be drawn and the type of drawing required. The **DRAWITEMSTRUCT** structure has the following form:

```
typedef struct tagDRAWITEMSTRUCT {  /* ditm */
    UINT  CtlType;
    UINT  CtlID;
    UINT  itemID;
    UINT  itemAction;
    UINT  itemState;
    HWND  hwndItem;
    HDC   hDC;
    RECT  rcItem;
    DWORD itemData;
} DRAWITEMSTRUCT;
```

Return Value

An application should return TRUE if it processes this message.

Comments

The **itemAction** member of the **DRAWITEMSTRUCT** structure defines the drawing operation that is to be performed. The data in this member allows the owner of the control to determine what drawing action is required.

Before returning from processing this message, an application should ensure that the device context identified by the *hDC* member of the **DRAWITEMSTRUCT** structure is in the default state.

Example

This example shows how to process the WM_DRAWITEM message:

```
LPDRAWITEMSTRUCT lpdis;

case WM_DRAWITEM:
    lpdis = (DRAWITEMSTRUCT FAR*) lParam;

    switch (lpdis->itemAction) {

        case ODA_DRAWENTIRE:
            .
            . /* Redraw the entire control or menu. */
            .
            return TRUE;

        case ODA_SELECT:
            .
            . /* Redraw to reflect current selection state. */
            .
            return TRUE;

        case ODA_FOCUS:
            .
            . /* Redraw to reflect current focus state. */
            .
            return TRUE;
    }
    break;
```

See Also

WM_COMPAREITEM, WM_DELETEITEM, WM_INITDIALOG, WM_MEASUREITEM

WM_DROPFILES

3.1

```
WM_DROPFILES
hDrop = (HANDLE) wParam;     /* handle of internal drop structure */
```

The WM_DROPFILES message is sent when the user releases the left mouse button over the window of an application that has registered itself as a recipient of dropped files.

Parameters

hDrop
Value of *wParam*. Identifies an internal data structure describing the dropped files. This handle is used by the **DragFinish**, **DragQueryFile**, and **DragQueryPoint** functions to retrieve information about the dropped files.

Return Value

An application should return zero if it processes this message.

See Also

DragAcceptFiles, **DragFinish**, **DragQueryFile**, **DragQueryPoint**

WM_ENABLE

2.x

```
WM_ENABLE
fEnabled = (BOOL) wParam;     /* the enabled/disabled flag */
```

The WM_ENABLE message is sent when an application changes the enabled state of a window. It is sent to the window whose enabled state is changing. This message is sent before the **EnableWindow** function returns but after the enabled state (WS_DISABLE style bit) of the window has changed.

Parameters

fEnabled
Value of *wParam*. Specifies whether the window has been enabled or disabled. This parameter is TRUE if the window has been enabled; it is FALSE if the window has been disabled.

Return Value

An application should return zero if it processes this message.

See Also

EnableWindow

WM_ENDSESSION

2.x

```
WM_ENDSESSION
fEndSession = (BOOL) wParam;     /* end-session flag */
```

The WM_ENDSESSION message is sent to an application that has returned a nonzero value in response to a WM_QUERYENDSESSION message. The WM_ENDSESSION message informs the application whether the session is actually ending.

Parameters

fEndSession
Value of *wParam*. Specifies whether the session is being ended. It is TRUE if the session is being ended; otherwise, it is FALSE.

Return Value

An application should return zero if it processes this message.

Comments

If the *fEndSession* parameter is TRUE, Windows can terminate any time after all applications have returned from processing this message. Therefore, an application should perform all tasks required for termination before returning from this message.

The application does not need to call the **DestroyWindow** or **PostQuitMessage** function when the session is ending.

See Also

DestroyWindow, **ExitWindows**, **PostQuitMessage**, WM_QUERYENDSESSION

WM_ENTERIDLE

2.x

```
WM_ENTERIDLE
fwSource = wParam;                  /* idle-source flag              */
hwndDlg = (HWND) LOWORD(lParam);    /* handle of dialog box or window */
```

The WM_ENTERIDLE message informs an application's main window procedure that a modal dialog box or a menu is entering an idle state. A modal dialog box or menu enters an idle state when no messages are waiting in its queue after it has processed one or more previous messages.

Parameters

fwSource
Value of *wParam*. Specifies whether the message is the result of a dialog box or a menu being displayed. This parameter can be one of the following values:

Value	Description
MSGF_DIALOGBOX	The system is idle because a dialog box is being displayed.
MSGF_MENU	The system is idle because a menu is being displayed.

hwndDlg
Value of the low-order word of *lParam*. Identifies the dialog box (if *fwSource* is MSGF_DIALOGBOX) or the handle of the window containing the displayed menu (if *fwSource* is MSGF_MENU).

Return Value

An application should return zero if it processes this message.

Comments

The **DefWindowProc** function returns zero when it processes this message.

See Also

DefWindowProc

WM_ERASEBKGND

2.x

```
WM_ERASEBKGND
hdc = (HDC) wParam; /* device-context handle */
```

The WM_ERASEBKGND message is sent when the window background needs to be erased (for example, when a window is resized). It is sent to prepare an invalidated region for painting.

Parameters

hdc
Value of *wParam*. Identifies the device context.

Return Value

An application should return nonzero if it erases the background; otherwise, it should return zero.

Comments

The **DefWindowProc** function erases the background by using the class background brush specified by the **hbrbackground** member of the **WNDCLASS** structure.

If the **hbrbackground** member is NULL, the application should process the WM_ERASEBKGND message and erase the background color. When processing the WM_ERASEBKGND message, the application must align the origin of the intended brush with the window coordinates by first calling the **UnrealizeObject** function for the brush and then selecting the brush.

Windows computes the background by using the MM_TEXT mapping mode. If the device context is using any other mapping mode, the area erased may not be within the visible part of the client area.

See Also **UnrealizeObject**, WM_ICONERASEBKGND

WM_FONTCHANGE

2.x

```
WM_FONTCHANGE
wParam = 0;      /* not used, must be zero */
lParam = 0L;     /* not used, must be zero */
```

An application sends the WM_FONTCHANGE message to all top-level windows in the system after changing the pool of font resources.

Parameters This message has no parameters.

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

Comments An application that adds or removes fonts from the system (for example, by using the **AddFontResource** or **RemoveFontResource** function) should send this message to all top-level windows.

To send the WM_FONTCHANGE message to all top-level windows, an application can call the **SendMessage** function with the *hwnd* parameter set to 0xFFFF.

See Also **AddFontResource**, **RemoveFontResource**, **SendMessage**

WM_GETDLGCODE

2.x

```
WM_GETDLGCODE
```

The WM_GETDLGCODE message is sent to the dialog box procedure associated with a control. Normally, Windows 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 This message has no parameters.

Return Value The return value is one or more of the following values, indicating which type of input the application processes:

Value	Meaning
DLGC_DEFPUSHBUTTON	Default push button
DLGC_HASSETSEL	EM_SETSEL messages
DLGC_PUSHBUTTON	Push button
DLGC_RADIOBUTTON	Radio button
DLGC_WANTALLKEYS	All keyboard input
DLGC_WANTARROWS	Arrow keys
DLGC_WANTCHARS	WM_CHAR messages
DLGC_WANTMESSAGE	All keyboard input (the application passes this message on to the control)
DLGC_WANTTAB	TAB key

Comments Although the **DefWindowProc** function always returns zero in response to the WM_GETDLGCODE message, the window procedures for the predefined control classes return a code appropriate for each class.

The WM_GETDLGCODE message and the returned values are useful only with user-defined dialog box controls or standard controls modified by subclassing.

WM_GETFONT

3.0

```
WM_GETFONT
wParam = 0;      /* not used, must be zero */
lParam = 0L;     /* not used, must be zero */
```

An application sends a WM_GETFONT message to a control to retrieve the font with which the control is currently drawing its text.

Parameters This message has no parameters.

Return Value The return value is the handle of the font used by the control, or it is NULL if the control is using the system font.

See Also WM_SETFONT

WM_GETMINMAXINFO

2.x

```
WM_GETMINMAXINFO
lpmmi = (MINMAXINFO FAR*) lParam; /* address of structure */
```

The WM_GETMINMAXINFO message is sent to a window whenever Windows needs the maximized position or dimensions of the window or needs the maximum or minimum tracking size of the window. The maximized size of a window is the size of the window when its borders are fully extended. The maximum tracking size of a window is the largest window size that can be achieved by using the borders to size the window. The minimum tracking size of a window is the smallest window size that can be achieved by using the borders to size the window.

Windows fills in a **MINMAXINFO** data structure, specifying default values for the various positions and dimensions. The application may change these values if it processes this message.

Parameters

lpmmi

Value of *lParam*. Points to a **MINMAXINFO** data structure. The **MINMAXINFO** structure has the following form:

```
typedef struct tagMINMAXINFO {  /* mmi */
    POINT ptReserved;
    POINT ptMaxSize;
    POINT ptMaxPosition;
    POINT ptMinTrackSize;
    POINT ptMaxTrackSize;
} MINMAXINFO;
```

Return Value

An application should return zero if it processes this message.

Example

This example processes a WM_GETMINMAXINFO message and sets the minimum tracking width of the window to 200 and the minimum tracking height of the window to 500:

```
MINMAXINFO FAR* lpmmi;

case WM_GETMINMAXINFO:
    lpmmi = (MINMAXINFO FAR*) lParam;
    lpmmi->ptMinTrackSize.x = 200;
    lpmmi->ptMinTrackSize.y = 500;

    break;
```

WM_GETTEXT

2.x

```
WM_GETTEXT
wParam = (WPARAM) cchTextMax;   /* number of bytes to copy    */
lParam = (LPARAM) lpszText;     /* address of buffer for text */
```

An application sends a WM_GETTEXT message to copy the text that corresponds to a window into a buffer provided by the caller.

Parameters

cchTextMax
Value of *wParam.* Specifies the maximum number of bytes to be copied, including the terminating null character.

lpszText
Value of *lParam.* Points to the buffer that is to receive the text.

Return Value

The return value is the number of bytes copied. It is CB_ERR if the message is sent to a combo box that has no edit control.

Comments

For an edit control, the text to be copied is the contents of the edit control. For a combo box, the text is the contents of the edit-control (or static-text) portion of the combo box. For a button, the text is the button name. For other windows, the text is the window title. To copy the text of an item in a list box, an application can use the LB_GETTEXT message.

When the WM_GETTEXT message is sent to a static control with the SS_ICON style, the handle of the icon will be returned in the first two bytes of the buffer pointed to by *lpszText.* This is true only if the WM_SETTEXT message has been used to set the icon.

Example

This example copies text from an edit control to a buffer:

```
HWND hwndMyEdit;
char szBuffer[32];

hwndMyEdit = GetDlgItem(hdlg, ID_MYEDITCONTROL);
SendMessage(hdlg, WM_GETTEXT, sizeof(szBuffer),
    (LPARAM) ((LPSTR) szBuffer));
```

See Also

LB_GETTEXT, WM_GETTEXTLENGTH, WM_SETTEXT

WM_GETTEXTLENGTH

2.x

```
WM_GETTEXTLENGTH
wParam = 0;      /* not used, must be zero */
lParam = 0L;     /* not used, must be zero */
```

An application sends a WM_GETTEXTLENGTH message to determine the length, in bytes, of the text associated with a window. The length does not include the terminating null character.

Parameters

This message has no parameters.

Return Value

The return value is a word specifying the length, in bytes, of the text.

Comments

For an edit control, the text to be copied is the contents of the edit control. For a combo box, the text is the contents of the edit-control (or static-text) portion of the combo box. For a button, the text is the button name. For other windows, the text is the window title. To determine the length of an item in a list box, an application can use the LB_GETTEXTLEN message.

Example

This example enables the push button in a dialog box if the user has entered text in an edit control in the dialog box:

```
case ID_MYEDITCONTROL:
    if (HIWORD(lParam) == EN_CHANGE)
        EnableWindow(GetDlgItem(hdlg, IDOK),
            (BOOL) SendMessage(LOWORD(lParam),
            WM_GETTEXTLENGTH, 0, 0L));
    return TRUE;
```

See Also

LB_GETTEXTLEN, WM_GETTEXT

WM_HSCROLL

2.x

```
WM_HSCROLL
wScrollCode = wParam;      /* scroll bar code                */
nPos = LOWORD(lParam);     /* current position of scroll box */
hwndCtl = (HWND) HIWORD(lParam); /* handle of the control    */
```

The WM_HSCROLL message is sent to a window when the user clicks the window's horizontal scroll bar.

Parameters

wScrollCode

Value of *wParam*. Specifies a scroll bar code that indicates the user's scrolling request. This parameter can be one of the following values:

Value	Description
SB_LEFT	Scroll to far left.
SB_LINELEFT	Scroll left.
SB_LINERIGHT	Scroll right.
SB_PAGELEFT	Scroll one page left.
SB_PAGERIGHT	Scroll one page right.
SB_RIGHT	Scroll to far right.
SB_THUMBPOSITION	Scroll to absolute position. The current position is specified by the *nPos* parameter.
SB_THUMBTRACK	Drag scroll box (thumb) to specified position. The current position is specified by the *nPos* parameter.

nPos

Value of the low-order word of *lParam*. Specifies the current position of the scroll box if the *wScrollCode* parameter is SB_THUMBPOSITION or SB_THUMBTRACK; otherwise, the *nPos* parameter is not used.

hwndCtl

Value of the high-order word of *lParam*. Identifies the control if WM_HSCROLL is sent by a scroll bar. If WM_HSCROLL is sent as a result of the user clicking a pop-up window's scroll bar, the high-order word is not used.

Return Value

An application should return zero if it processes this message.

Comments

The SB_THUMBTRACK scroll bar code typically is used by applications that give some feedback while the scroll box is being dragged.

If an application scrolls the contents of the window, it must also reset the position of the scroll box by using the **SetScrollPos** function.

See Also

SetScrollPos, WM_VSCROLL

WM_HSCROLLCLIPBOARD

2.x

```
WM_HSCROLLCLIPBOARD
hwndCBViewer = (HWND) wParam;    /* handle of clipboard viewer */
wScrollCode = LOWORD(lParam);    /* scroll bar code            */
nPos = (int) HIWORD(lParam);     /* scroll box position        */
```

The WM_HSCROLLCLIPBOARD message is sent by the clipboard viewer to the clipboard owner when the clipboard data has the CF_OWNERDISPLAY format and an event occurs in the clipboard viewer's horizontal scroll bar. The owner should scroll the clipboard image, invalidate the appropriate section, and update the scroll bar values.

Parameters

hwndCBViewer
Value of *wParam*. Identifies a clipboard-viewer window.

wScrollCode
Value of the low-order word of *lParam*. Specifies a scroll bar code. This parameter can be one of the following values:

Value	Description
SB_BOTTOM	Scroll to lower right.
SB_ENDSCROLL	End scroll.
SB_LINEDOWN	Scroll one line down.
SB_LINEUP	Scroll one line up.
SB_PAGEDOWN	Scroll one page down.
SB_PAGEUP	Scroll one page up.
SB_THUMBPOSITION	Scroll to absolute position.
SB_TOP	Scroll to upper left.

nPos
Value of the high-order word of *lParam*. Specifies the scroll box position if the scroll bar code is SB_THUMBPOSITION; otherwise, the high-order word of *lParam* is not used.

Return Value

An application should return zero if it processes this message.

Comments

The clipboard owner should use the **InvalidateRect** function or repaint as needed. The scroll bar position should also be reset.

See Also

InvalidateRect, WM_VSCROLLCLIPBOARD

WM_ICONERASEBKGND

3.0

```
WM_ICONERASEBKGND
hdc = (HDC) wParam;      /* device-context handle */
```

The WM_ICONERASEBKGND message is sent to a minimized (iconic) window when the background of the icon must be filled before painting the icon. A window receives this message only if a class icon is defined for the window; otherwise, WM_ERASEBKGND is sent.

Parameters

hdc
Value of *wParam*. Identifies the device context of the icon.

Return Value

An application should return zero if it processes this message.

Comments

The **DefWindowProc** function fills the icon background with the background brush of the parent window.

See Also

DefWindowProc, WM_ERASEBKGND

WM_INITDIALOG

2.x

```
WM_INITDIALOG
hwndFocus = (HWND) wParam;  /* handle of control for focus */
dwData = lParam;            /* application-specific data   */
```

The WM_INITDIALOG message is sent to a dialog box procedure immediately before the dialog box is displayed.

Parameters

hwndFocus
Value of *wParam*. Identifies the first control in the dialog box that can be given the input focus. Usually, this is the first control in the dialog box with the WS_TABSTOP style.

dwData
Value of *lParam*. Specifies application-specific data that was passed by the function used to create the dialog box if the dialog box was created by one of the following functions:

CreateDialogParam
DialogBoxIndirectParam
DialogBoxParam

Return Value

An application should return nonzero to set the input focus to the control identified by the *hwndFocus* parameter. An application should return zero if the dialog box procedure uses the **SetFocus** function to set the input focus to a different control in the dialog box.

Example

This example changes the font used by controls in a dialog box to a font that is not bold.

```
HFONT hDlgFont;
LOGFONT lFont;

case WM_INITDIALOG:

    /* Get dialog box font and create version that is not bold. */

    hDlgFont = (HFONT) NULL;
    if ((hDlgFont = (HFONT) SendMessage(hdlg, WM_GETFONT, 0, 0L))) {
        if (GetObject(hDlgFont, sizeof(LOGFONT), (LPSTR) &lFont)) {
            lFont.lfWeight = FW_NORMAL;
            if (hDlgFont = CreateFontIndirect((LPLOGFONT) &lFont)) {
                SendDlgItemMessage(hdlg, ID_CTRL1, WM_SETFONT,
                    hDlgFont, 0L);
                SendDlgItemMessage(hdlg, ID_CTRL2, WM_SETFONT,
                    hDlgFont, 0L);
                .
                . /* Set font for remaining controls. */
                .
            }
        }
    }
    return TRUE;
```

See Also

CreateDialogParam, DialogBoxIndirectParam, DialogBoxParam, SetFocus

WM_INITMENU

2.x

```
WM_INITMENU
hmenuInit = (HMENU) wParam; /* handle of menu to initialize */
```

The WM_INITMENU message is sent when a menu is about to become active. It occurs when the user clicks an item on the menu bar or presses a menu key. This allows an application to modify the menu before it is displayed.

Parameters

hmenuInit
Value of *wParam*. Identifies the menu to be initialized.

Return Value

An application should return zero if it processes this message.

Comments

This message is sent only when a menu is first accessed; only one WM_INITMENU message is generated for each access. This means, for example, that moving the mouse across several menu items while holding down the button does not generate new messages. WM_INITMENU does not provide information about menu items.

See Also

WM_INITMENUPOPUP

WM_INITMENUPOPUP

2.x

```
WM_INITMENUPOPUP
hmenuPopup = (HMENU) wParam;          /* handle of pop-up menu */
nIndex = (int) LOWORD(lParam);        /* index of pop-up menu  */
fSystemMenu = (BOOL) HIWORD(lParam); /* System-menu flag      */
```

The WM_INITMENUPOPUP message is sent when a pop-up menu is about to become active. This allows an application to modify the pop-up menu before it is displayed, without changing the entire menu.

Parameters

hmenuPopup
Value of *wParam*. Identifies the pop-up menu.

nIndex
Value of the low-order word of *lParam*. Specifies the index of the pop-up menu in the main menu.

fSystemMenu
Value of the high-order word of *lParam*. Specifies a nonzero value if the pop-up menu is the System menu (sometimes referred to as the Control menu); otherwise, this parameter is zero.

Return Value

An application should return zero if it processes this message.

Example

This example initializes the items in a pop-up menu:

```
int  nCount;
WORD wItem;
UINT uID;

case WM_INITMENUPOPUP:
    nCount = GetMenuItemCount(wParam);
    for (wItem = 0; wItem < nCount; wItem++) {
        uID = GetMenuItemID(wParam, wItem);
        .
        . /* Initialize menu items. */
        .
    }
    break;
```

See Also

WM_INITMENU

WM_KEYDOWN

2.x

```
WM_KEYDOWN
wVkey = wParam;          /* virtual-key code */
dwKeyData = lParam;      /* key data         */
```

The WM_KEYDOWN message is sent when a nonsystem key is pressed. A nonsystem key is a key that is pressed when the ALT key is *not* pressed, or it is a key that is pressed when a window has the input focus.

Parameters

wVkey

Value of *wParam*. Specifies the virtual-key code of the given key.

dwKeydata

Value of *lParam*. Specifies the repeat count, scan code, extended key, context code, previous key state, and key-transition state, as shown in the following table:

Bit	Description
0–15	Specifies the repeat count. The value is the number of times the keystroke is repeated as a result of the user holding down the key.
16–23	Specifies the scan code. The value depends on the original equipment manufacturer (OEM).
24	Specifies whether the key is an extended key, such as a function key or a key on the numeric keypad. The value is 1 if it is an extended key; otherwise, it is 0.

Bit	Description
25–26	Not used.
27–28	Used internally by Windows.
29	Specifies the context code. The value is 1 if the ALT key is held down while the key is pressed; otherwise, the value is 0.
30	Specifies the previous key state. The value is 1 if the key is down before the message is sent, or it is 0 if the key is up.
31	Specifies the key-transition state. The value is 1 if the key is being released, or it is 0 if the key is being pressed.

For a WM_KEYDOWN message, the value of bit 29 (context code) is 0 and the value of bit 31 (key-transition state) is 0.

Return Value

An application should return zero if it processes this message.

Comments

Because of the autorepeat feature, more than one WM_KEYDOWN message may occur before a WM_KEYUP message is sent. The previous key state (bit 30) can be used to determine whether the WM_KEYDOWN message indicates the first down transition or a repeated down transition.

For IBM Enhanced 101- and 102-key keyboards, enhanced keys are the right ALT key and the right CTRL key on the main section of the keyboard; the INS, DEL, HOME, END, PAGE UP, PAGE DOWN, and arrow keys in the clusters to the left of the numeric keypad; and the division (/) and ENTER key on the numeric keypad. Some other keyboards may support the extended-key bit in the *dwKeyData* parameter.

See Also

WM_CHAR, WM_KEYUP

WM_KEYUP

2.x

```
WM_KEYUP
wVkey = wParam;          /* virtual-key code */
dwKeyData = lParam;      /* key data         */
```

The WM_KEYUP message is sent when a nonsystem key is released. A nonsystem key is a key that is pressed when the ALT key is *not* pressed, or it is a key that is pressed when a window has the input focus.

Parameters

wVkey

Value of *wParam*. Specifies the virtual-key code of the given key.

dwKeyData

Value of *lParam*. Specifies the repeat count, scan code, extended key, context code, previous key state, and key-transition state, as shown in the following table:

Bit	Description
0–15	Specifies the repeat count. The value is the number of times the keystroke is repeated as a result of the user holding down the key.
16–23	Specifies the scan code. The value depends on the original equipment manufacturer (OEM).
24	Specifies whether the key is an extended key, such as a function key or a key on the numeric keypad. The value is 1 if it is an extended key; otherwise, it is 0.
25–26	Not used.
27–28	Used internally by Windows.
29	Specifies the context code. The value is 1 if the ALT key is held down while the key is pressed; otherwise, the value is 0.
30	Specifies the previous key state. The value is 1 if the key is down before the message is sent, or it is 0 if the key is up.
31	Specifies the key-transition state. The value is 1 if the key is being released, or it is 0 if the key is being pressed.

For a WM_KEYUP message, the value of bit 29 (context code) is 0 and the value of bit 31 (key-transition state) is 1.

Return Value

An application should return zero if it processes this message.

Comments

For IBM Enhanced 101- and 102-key keyboards, enhanced keys are the right ALT key and the right CTRL key on the main section of the keyboard; the INS, DEL, HOME, END, PAGE UP, PAGE DOWN, and arrow keys in the clusters to the left of the numeric keypad; and the division (/) and ENTER keys on the numeric keypad. Some other keyboards may support the extended-key bit in the *dwKeyData* parameter.

See Also

WM_CHAR, WM_KEYDOWN

WM_KILLFOCUS

2.x

```
WM_KILLFOCUS
hwndGetFocus = (HWND) lParam;   /* handle of window receiving focus */
```

The WM_KILLFOCUS message is sent immediately before a window loses the input focus.

Parameters

hwndGetFocus
Value of *wParam*. Identifies the window that receives the input focus. (This parameter may be NULL.)

Return Value

An application should return zero if it processes this message.

Comments

If an application is displaying a caret, the caret should be destroyed at this point.

See Also

SetFocus, WM_SETFOCUS

WM_LBUTTONDBLCLK

2.x

```
WM_LBUTTONDBLCLK
fwKeys = wParam;          /* key flags                      */
xPos = LOWORD(lParam);    /* horizontal position of cursor */
yPos = HIWORD(lParam);    /* vertical position of cursor   */
```

The WM_LBUTTONDBLCLK message is sent when the user double-clicks the left mouse button.

Parameters

fwKeys
Value of *wParam*. Indicates whether various virtual keys are down. This parameter can be any combination of the following values:

Value	Description
MK_CONTROL	Set if CTRL key is down.
MK_LBUTTON	Set if left button is down.
MK_MBUTTON	Set if middle button is down.
MK_RBUTTON	Set if right button is down.
MK_SHIFT	Set if SHIFT key is down.

xPos
Value of the low-order word of *lParam*. Specifies the x-coordinate of the cursor. The coordinate is relative to the upper-left corner of the window.

yPos
Value of the high-order word of *lParam*. Specifies the y-coordinate of the cursor. The coordinate is relative to the upper-left corner of the window.

Return Value

An application should return zero if it processes this message.

Comments

Only windows that have the CS_DBLCLKS class style can receive WM_LBUTTONDBLCLK messages. Windows generates a WM_LBUTTONDBLCLK message when the user presses, releases, and again presses the left mouse button within the system's double-click time limit. Double-clicking the left mouse button actually generates four messages: a WM_LBUTTONDOWN message, a WM_LBUTTONUP message, the WM_LBUTTONDBLCLK message, and another WM_LBUTTONUP message.

See Also

WM_LBUTTONDOWN, WM_LBUTTONUP

WM_LBUTTONDOWN

2.x

```
WM_LBUTTONDOWN
fwKeys = wParam;          /* key flags                        */
xPos = LOWORD(lParam);    /* horizontal position of cursor */
yPos = HIWORD(lParam);    /* vertical position of cursor   */
```

The WM_LBUTTONDOWN message is sent when the user presses the left mouse button.

Parameters

fwKeys
Value of *wParam*. Specifies whether various virtual keys are down. This parameter can be any combination of the following values:

Value	Description
MK_CONTROL	Set if CTRL key is down.
MK_MBUTTON	Set if middle button is down.
MK_RBUTTON	Set if right button is down.
MK_SHIFT	Set if SHIFT key is down.

xPos
Value of the low-order word of *lParam*. Specifies the x-coordinate of the cursor. The coordinate is relative to the upper-left corner of the window.

yPos
Value of the high-order word of *lParam*. Specifies the y-coordinate of the cursor. The coordinate is relative to the upper-left corner of the window.

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

See Also WM_LBUTTONDBLCLK, WM_LBUTTONUP

WM_LBUTTONUP

2.x

```
WM_LBUTTONUP
fwKeys = wParam;          /* key flags                        */
xPos = LOWORD(lParam);    /* horizontal position of cursor */
yPos = HIWORD(lParam);    /* vertical position of cursor   */
```

The WM_LBUTTONUP message is sent when the user releases the left mouse button.

Parameters

fwKeys
Value of *wParam*. Indicates whether various virtual keys are down. This parameter can be any combination of the following values:

Value	Description
MK_CONTROL	Set if CTRL key is down.
MK_MBUTTON	Set if middle button is down.
MK_RBUTTON	Set if right button is down.
MK_SHIFT	Set if SHIFT key is down.

xPos
Value of the low-order word of *lParam*. Specifies the x-coordinate of the cursor. The coordinate is relative to the upper-left corner of the window.

yPos
Value of the high-order word of *lParam*. Specifies the y-coordinate of the cursor. The coordinate is relative to the upper-left corner of the window.

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

See Also WM_LBUTTONDBLCLK, WM_LBUTTONDOWN

WM_MBUTTONDBLCLK

2.x

```
WM_MBUTTONDBLCLK
fwKeys = wParam;          /* key flags                        */
xPos = LOWORD(lParam);    /* horizontal position of cursor */
yPos = HIWORD(lParam);    /* vertical position of cursor   */
```

The WM_MBUTTONDBLCLK message is sent when the user double-clicks the middle mouse button.

Parameters

fwKeys

Value of *wParam*. Indicates whether various virtual keys are down. This parameter can be any combination of the following values:

Value	Description
MK_CONTROL	Set if CTRL key is down.
MK_LBUTTON	Set if left button is down.
MK_MBUTTON	Set if middle button is down.
MK_RBUTTON	Set if right button is down.
MK_SHIFT	Set if SHIFT key is down.

xPos

Value of the low-order word of *lParam*. Specifies the x-coordinate of the cursor. The coordinate is relative to the upper-left corner of the window.

yPos

Value of the high-order word of *lParam*. Specifies the y-coordinate of the cursor. The coordinate is relative to the upper-left corner of the window.

Return Value

An application should return zero if it processes this message.

Comments

Only windows that have the CS_DBLCLKS class style can receive WM_MBUTTONDBLCLK messages. Windows generates a WM_MBUTTONDBLCLK message when the user presses, releases, and again presses the middle mouse button within the system's double-click time limit. Double-clicking the middle mouse button actually generates four messages: a WM_MBUTTONDOWN message, a WM_MBUTTONUP message, the WM_MBUTTONDBLCLK message, and another WM_MBUTTONUP message.

See Also

WM_MBUTTONDOWN, WM_MBUTTONUP

WM_MBUTTONDOWN

2.x

```
WM_MBUTTONDOWN
fwKeys = wParam;          /* key flags                        */
xPos = LOWORD(lParam);    /* horizontal position of cursor */
yPos = HIWORD(lParam);    /* vertical position of cursor   */
```

The WM_MBUTTONDOWN message is sent when the user presses the middle mouse button.

Parameters

fwKeys
Value of *wParam*. Indicates whether various virtual keys are down. This parameter can be any combination of the following values:

Value	Description
MK_CONTROL	Set if CTRL key is down.
MK_LBUTTON	Set if left button is down.
MK_RBUTTON	Set if right button is down.
MK_SHIFT	Set if SHIFT key is down.

xPos
Value of the low-order word of *lParam*. Specifies the x-coordinate of the cursor. The coordinate is relative to the upper-left corner of the window.

yPos
Value of the high-order word of *lParam*. Specifies the y-coordinate of the cursor. The coordinate is relative to the upper-left corner of the window.

Return Value

An application should return zero if it processes this message.

See Also

WM_MBUTTONDBLCLK, WM_MBUTTONUP

WM_MBUTTONUP

2.x

```
WM_MBUTTONUP
fwKeys = wParam;          /* key flags                        */
xPos = LOWORD(lParam);    /* horizontal position of cursor */
yPos = HIWORD(lParam);    /* vertical position of cursor   */
```

The WM_MBUTTONUP message is sent when the user releases the middle mouse button.

Parameters

fwKeys

Value of *wParam*. Indicates whether various virtual keys are down. This parameter can be any combination of the following values:

Value	Description
MK_CONTROL	Set if CTRL key is down.
MK_LBUTTON	Set if left button is down.
MK_RBUTTON	Set if right button is down.
MK_SHIFT	Set if SHIFT key is down.

xPos

Value of the low-order word of *lParam*. Specifies the x-coordinate of the cursor. The coordinate is relative to the upper-left corner of the window.

yPos

Value of the high-order word of *lParam*. Specifies the y-coordinate of the cursor. The coordinate is relative to the upper-left corner of the window.

Return Value

An application should return zero if it processes this message.

See Also

WM_MBUTTONDBLCLK, WM_MBUTTONDOWN

WM_MDIACTIVATE

3.0

```
WM_MDIACTIVATE
/* Message sent to MDI client */
wParam =  (WPARAM) (HWND) hwndChildAct; /* child to activate       */
lParam = 0L;                            /* not used, must be zero  */

/* Message received by MDI child */
wParam = (WPARAM) fActivate;            /* activation flag         */
hwndAct = (HWND) LOWORD(lParam);        /* child being activated   */
hwndDeact = (HWND) HIWORD(lParam);      /* child being deactivated */
```

An application sends the WM_MDIACTIVATE message to a multiple document interface (MDI) client window to instruct the client window to activate a different MDI child window. As the client window processes this message, it sends WM_MDIACTIVATE to the child window being deactivated and to the child window being activated.

Parameters

In message sent to MDI client window:

hwndChildAct
Value of *wParam*. Identifies the MDI child window to be activated.

In message received by MDI child window:

fActivate
Value of *wParam*. Specifies whether to activate or deactivate the child window. If this parameter is TRUE, the child window is activated. If this parameter is FALSE, the child window is deactivated.

hwndAct
Value of the low-order word of *lParam*. Identifies the child window being activated.

hwndDeact
Value of the high-order word of *lParam*. Identifies the child window being deactivated.

Return Value

An application should return zero if it processes this message.

Comments

An MDI child window is activated independently of the MDI frame window. When the frame window becomes active, the child window that was last activated with the WM_MDIACTIVATE message receives the WM_NCACTIVATE message to draw an active window frame and title bar; it does not receive another WM_MDIACTIVATE message.

See Also

WM_MDIGETACTIVE, WM_NCACTIVATE, WM_MDINEXT

WM_MDICASCADE

3.0

```
WM_MDICASCADE
fnCascade = wParam;      /* cascade flag */
```

The WM_MDICASCADE message is sent to a multiple document interface (MDI) client window to arrange all its child windows in a cascade format.

Parameters

fnCascade
Value of *wParam*. Specifies a cascade flag. Currently, only the following flag may be specified:

Value	Meaning
MDITILE_SKIPDISABLED	Prevents disabled MDI child windows from being cascaded.

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

See Also WM_MDIICONARRANGE, WM_MDITILE

WM_MDICREATE

3.0

```
WM_MDICREATE
wParam = 0;                                    /* not used, must be zero */
lParam = (LPARAM) (MDICREATESTRUCT FAR*) lpmcs; /* structure address  */
```

An application sends the WM_MDICREATE message to a multiple document interface (MDI) client window to create a child window.

Parameters *lpmcs*

Value of *lParam*. Points to an **MDICREATESTRUCT** structure. The **MDICREATESTRUCT** structure has the following form:

```
typedef struct tagMDICREATESTRUCT {     /* mdic */
    LPCSTR    szClass;
    LPCSTR    szTitle;
    HINSTANCE hOwner;
    int       x;
    int       y;
    int       cx;
    int       cy;
    DWORD     style;
    LPARAM    lParam;
} MDICREATESTRUCT;
```

Return Value The return value is the handle of the new window in the low-order word and zero in the high-order word.

Comments The window is created with the style bits WS_CHILD, WS_CLIPSIBLINGS, WS_CLIPCHILDREN, WS_SYSMENU, WS_CAPTION, WS_THICKFRAME, WS_MINIMIZEBOX, and WS_MAXIMIZEBOX, plus additional style bits specified in the **MDICREATESTRUCT** structure to which *lpmcs* points.

Windows adds the title of the new child window to the window menu of the frame window. An application should create all child windows of the client window with this message.

If the MDIS_ALLCHILDSTYLES style is set when the MDI client window is created, **CreateWindow** overrides the default style bits.

If a client window receives any message that changes the activation of child windows while the currently active MDI child window is maximized, Windows restores the currently active child window and maximizes the newly activated child window.

When the MDI child window is created, Windows sends the WM_CREATE message to the window. The *lpmcs* parameter of the WM_CREATE message contains a pointer to a **CREATESTRUCT** structure. The **lpCreateParams** member of the **CREATESTRUCT** structure contains a pointer to the **MDICREATESTRUCT** structure passed with the WM_MDICREATE message that created the MDI child window.

An application should not send a second WM_MDICREATE message while a WM_MDICREATE message is still being processed. For example, it should not send a WM_MDICREATE message while an MDI child window is processing its WM_CREATE message.

See Also WM_MDIDESTROY

WM_MDIDESTROY

3.0

```
WM_MDIDESTROY
hwndChild = (HWND) wParam;  /* handle of child to destroy */
```

An application sends the WM_MDIDESTROY message to a multiple document interface (MDI) client window to close an MDI child window.

Parameters

hwndChild
Value of *wParam*. Identifies the child window to destroy.

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

Comments This message removes the title of the child window from the frame window and deactivates the child window. An application should close all MDI child windows with this message.

If a client window receives any message that changes the activation of child windows while the currently active MDI child window is maximized, Windows restores the currently active child window and maximizes the newly activated child window.

See Also WM_MDICREATE

WM_MDIGETACTIVE

3.0

```
WM_MDIGETACTIVE
```

The WM_MDIGETACTIVE message retrieves the multiple document interface (MDI) child window that is active, along with a flag indicating whether the child window is maximized.

Parameters This message has no parameters.

Return Value The return value is the handle of the active MDI child window in its low-order word. If the window is maximized, the high-order word is 1; otherwise, the high-order word is 0.

See Also WM_MDIACTIVATE

WM_MDIICONARRANGE

3.0

```
WM_MDIICONARRANGE
```

The WM_MDIICONARRANGE message is sent to a multiple document interface (MDI) client window to arrange all minimized document child windows. It does not affect child windows that are not minimized.

Parameters This message has no parameters.

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

See Also WM_MDICASCADE, WM_MDITILE

WM_MDIMAXIMIZE

3.0

```
WM_MDIMAXIMIZE
hwndMaximize = (HWND) wParam;    /* handle of child to maximize */
```

The WM_MDIMAXIMIZE message causes a multiple document interface (MDI) client window to maximize an MDI child window. When a child window is maximized, Windows resizes it to make its client area fill the client window. Windows places the child window's System menu (sometimes referred to as the Control menu) in the frame's menu bar so that the user can restore or minimize the child window; Windows adds the title of the child window to the frame window's menu of child windows.

Parameters

hwndMaximize
Value of *wParam*. Identifies the child window to maximize.

Return Value

An application should return zero if it processes this message.

Comments

If an MDI client window receives any message that changes the activation of its child windows while the currently active MDI child window is maximized, Windows restores the currently active child window and maximizes the newly activated child window.

WM_MDINEXT

3.0

```
WM_MDINEXT
wParam = (WPARAM) hwndChild; /* handle of child window        */
lParam = (LPARAM) fNext;     /* next or previous child window */
```

An application sends the WM_MDINEXT message to a multiple document interface (MDI) client window to activate the child window immediately behind the currently active child window and place the currently active child window behind all other child windows.

Parameters

hwndChild
Value of *wParam*. Specifies the handle of the child window.

fNext
Value of *lParam*. If this parameter is zero, the message specifies that the next MDI child window should be activated. If this parameter is nonzero, the message specifies that the previous MDI child window should be activated.

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

Comments If an MDI client window receives any message that changes the activation of its child windows while the currently active MDI child window is maximized, Windows restores the currently active child window and maximizes the newly activated child window.

See Also WM_MDIACTIVATE, WM_MDIGETACTIVE

WM_MDIRESTORE

3.0

```
WM_MDIRESTORE
wParam = (WPARAM) wIDChild;  /* handle of child window */
```

An application sends the WM_MDIRESTORE message to a multiple document interface (MDI) client window to restore an MDI child window from maximized or minimized size.

Parameters *wIDChild*
Value of *wParam*. Specifies the handle of the child window.

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

See Also WM_MDIMAXIMIZE

WM_MDISETMENU

3.0

```
WM_MDISETMENU
wParam = (WPARAM) (BOOL) fRefresh;              /* refresh flag */
lParam = MAKELPARAM(hmenuFrame, hmenuWindow);  /* new menus    */
```

An application sends a WM_MDISETMENU message to replace the menu of a multiple document interface (MDI) frame window, the Window pop-up menu, or both.

Parameters *fRefresh*
Value of *wParam*. Specifies whether to refresh the current menus or specify new menus. It is TRUE if the menus should just be refreshed. It is FALSE if,

instead, the *hmenuFrame* and *hmenuWindow* parameters should be used to specify new menus for the window.

hmenuFrame
Value of the low-order word of *lParam*. Identifies the new frame-window menu. If this parameter is zero, the frame-window menu is not changed.

hmenuWindow
Value of the high-order word of *lParam*. Identifies the new Window pop-up menu. If this parameter is zero, the Window pop-up menu is not changed.

Return Value

The return value is the handle of the frame-window menu replaced by this message.

Comments

After sending this message, an application must call the **DrawMenuBar** function to update the menu bar.

If this message replaces the Window pop-up menu, MDI child-window menu items are removed from the previous Window menu and added to the new Window pop-up menu.

If an MDI child window is maximized and this message replaces the MDI frame-window menu, the System menu (sometimes referred to as the Control menu) and restore controls are removed from the previous frame-window menu and added to the new menu.

See Also

DrawMenuBar

WM_MDITILE

3.0

```
WM_MDITILE
fTile = wParam;      /* tiling flag */
```

The WM_MDITILE message is sent to a multiple document interface (MDI) client window to arrange all its child windows in a tiled format.

Parameters

fTile
Value of *wParam*. Specifies a tiling flag. This parameter can be one of the following flags:

Value	Meaning
MDITILE_HORIZONTAL	Tiles MDI child windows so that they are wide rather than tall.
MDITILE_SKIPDISABLED	Prevents disabled MDI child windows from being tiled.
MDITILE_VERTICAL	Tiles MDI child windows so that they are tall rather than wide.

Return Value

An application should return zero if it processes this message.

See Also

WM_MDICASCADE, WM_MDIICONARRANGE

WM_MEASUREITEM

3.0

```
WM_MEASUREITEM
nIDCtl = (int) wParam;                      /* control identifier   */
lpmisCtl = (MEASUREITEMSTRUCT FAR*) lParam; /* address of structure */
```

The WM_MEASUREITEM message is sent to the owner of an owner-drawn button, combo box, list box, or menu item when the control is created. When the owner receives the message, the owner fills in the **MEASUREITEMSTRUCT** structure pointed to by the *lpmisCtl* message parameter and returns; this informs Windows of the dimensions of the control. If a list box or combo box is created with the LBS_OWNERDRAWVARIABLE or CBS_OWNERDRAWVARIABLE style, this message is sent to the owner for each item in the control; otherwise, this message is sent once.

Parameters

nIDCtl

Value of *wParam*. Specifies the identifier of the control that sent the WM_MEASUREITEM message. This parameter is 0 if the message was sent by a menu. This parameter is –1 when the system is requesting the dimensions of an edit control in an owner-drawn combo box.

lpmisCtl

Value of *lParam*. Points to a **MEASUREITEMSTRUCT** structure that contains the dimensions of the owner-drawn control.

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

```
typedef struct tagMEASUREITEMSTRUCT {   /* mi */
    UINT   CtlType;
    UINT   CtlID;
    UINT   itemID;
    UINT   itemWidth;
    UINT   itemHeight;
    DWORD  itemData;
} MEASUREITEMSTRUCT;
```

Return Value

An application should return TRUE if it processes this message.

Comments

Windows sends the WM_MEASUREITEM message to the owner of a combo box or list box created with the OWNERDRAWFIXED style before sending WM_INITDIALOG. As a result, when the owner receives this message, Windows has not yet determined the height and width of the font used in the control; function calls and calculations requiring these values should occur in the main function of the application or library.

See Also

WM_COMPAREITEM, WM_DELETEITEM, WM_DRAWITEM, WM_INITDIALOG

WM_MENUCHAR

2.x

```
WM_MENUCHAR
chUser = wParam;                    /* ASCII character    */
fMenu = LOWORD(lParam);             /* menu flag          */
hmenu = (HMENU) HIWORD(lParam);     /* handle of the menu */
```

The WM_MENUCHAR message is sent when the user presses the key corresponding to a menu mnemonic character that doesn't match any of the predefined mnemonics in the current menu. It is sent to the window that owns the menu.

Parameters

chUser

Value of *wParam*. Specifies the ASCII character that corresponds to the key the user pressed.

fMenu

Value of the low-order word of *lParam*. Specifies the type of the selected menu. This parameter can be one of the following values:

Value	Meaning
MF_POPUP	The menu is a pop-up menu.
MF_SYSMENU	The menu is a System menu (sometimes referred to as a Control menu).

hmenu
Value of the high-order word of *lParam*. Identifies the selected menu.

Return Value

The return value is one of the following command code values in the high-order word:

Value	Description
0	Informs Windows that it should discard the character corresponding to the key the user pressed, and creates a short beep on the system speaker.
1	Informs Windows that it should close the current menu.
2	Informs Windows that the low-order word of the return value contains the item number for a specific item. This item is selected by Windows.

The low-order word is ignored if the high-order word contains 0 or 1. An application should process this message when an accelerator key has been used to select a bitmap placed in a menu.

Comments

The WM_MENUCHAR message is generated when the user presses ALT and any key, even if the key does not correspond to a mnemonic character. In this case, the *hmenu* parameter contains the window handle of the menu.

WM_MENUSELECT

2.x

```
WM_MENUSELECT
wIDItem = wParam;               /* item identifier or menu handle */
fwMenu = LOWORD(lParam);        /* menu flags                     */
hmenu = (HMENU) HIWORD(lParam); /* handle of the menu             */
```

The WM_MENUSELECT message is sent to the window associated with a menu when the user selects a menu item.

Parameters

wIDItem
Value of *wParam*. Specifies the menu-item identifier if the selected item is a menu item. If the selected item contains a pop-up menu, *wIDItem* contains the handle of the pop-up menu.

fwMenu
Low word of *lParam*. Specifies one or more menu flags. This parameter can be a combination of the following values:

Flag	Description
MF_BITMAP	Item is a bitmap.
MF_CHECKED	Item is checked.
MF_DISABLED	Item is disabled.
MF_GRAYED	Item is grayed.
MF_MOUSESELECT	Item was selected with a mouse.
MF_OWNERDRAW	Item is an owner-drawn item.
MF_POPUP	Item contains a pop-up menu.
MF_SEPARATOR	Item is a menu-item separator.
MF_SYSMENU	Item is contained in the System menu (sometimes referred to as the Control menu). The *hmenu* parameter identifies the System menu associated with the message.

hmenu
High word of *lParam*. If the *fwMenu* parameter contains the MF_SYSMENU flag, this parameter specifies the menu handle of the System menu.

Return Value

An application should return zero if it processes this message.

Comments

If the *fwMenu* parameter contains –1 and the *hmenu* parameter contains 0, Windows has closed the menu. This occurs both when the menu is closed because the user pressed ESC or clicked outside the menu and when the user has selected a menu item.

WM_MOUSEACTIVATE

2.x

```
WM_MOUSEACTIVATE
hwndTopLevel = (HWND) wParam;    /* handle of top-level parent */
wHitTestCode = LOWORD(lParam);   /* hit-test code              */
wMsg = HIWORD(lParam);           /* mouse-message identifier   */
```

The WM_MOUSEACTIVATE message is sent when the cursor is in an inactive window and the user presses a mouse button. The parent window receives this message only if the child window passes it to the **DefWindowProc** function.

Parameters

hwndTopLevel
Value of *wParam*. Identifies the top-level parent window of the window being activated.

wHitTestCode
Value of the low-order word of *lParam*. Specifies the hit-test area code. A hit test is a test that determines the location of the cursor.

wMsg
Value of the high-order word of *lParam*. Specifies the identifier of the mouse message.

Return Value

The return value specifies whether the window should be activated and whether the mouse event should be discarded. It must be one of the following values:

Value	Meaning
MA_ACTIVATE	Activate the window.
MA_NOACTIVATE	Do not activate the window.
MA_ACTIVATEANDEAT	Activate the window and discard the mouse event.
MA_NOACTIVATEANDEAT	Do not activate the window; discard the mouse event.

Comments

If the child window passes the message to the **DefWindowProc** function, **DefWindowProc** passes this message to a window's parent window before any processing occurs. If the parent window returns a nonzero value, processing is halted.

WM_MOUSEMOVE

2.x

```
WM_MOUSEMOVE
fwKeys = wParam;          /* key flags                        */
xPos = LOWORD(lParam);    /* horizontal position of cursor */
yPos = HIWORD(lParam);    /* vertical position of cursor   */
```

The WM_MOUSEMOVE message is sent to a window when the mouse cursor moves. If the mouse is not captured, the message goes to the window beneath the cursor. Otherwise, the message goes to the window that has captured the mouse.

Parameters

fwKeys
Value of *wParam*. Indicates whether various virtual keys are down. This parameter can be any combination of the following values:

Value	Description
MK_CONTROL	Set if CTRL key is down.
MK_LBUTTON	Set if left button is down.
MK_MBUTTON	Set if middle button is down.
MK_RBUTTON	Set if right button is down.
MK_SHIFT	Set if SHIFT key is down.

xPos
Value of the low-order word of *lParam*. Specifies the x-coordinate of the cursor, as a screen coordinate.

yPos
Value of the high-order word of *lParam*. Specifies the y-coordinate of the cursor, as a screen coordinate.

Return Value

An application should return zero if it processes this message.

Comments

The **MAKEPOINT** macro can be used to convert the *lParam* parameter to a **POINT** structure.

See Also

SetCapture, WM_NCHITTEST

WM_MOVE

2.x

```
WM_MOVE
xPos = (int) LOWORD(lParam);     /* horizontal position */
yPos = (int) HIWORD(lParam);     /* vertical position   */
```

The WM_MOVE message is sent after a window has been moved.

Parameters

xPos
Value of the low-order word of *lParam*. Specifies the new x-coordinate of the upper-left corner of the client area of the window.

yPos
Value of the high-order word of *lParam*. Specifies the new y-coordinate of the upper-left corner of the client area of the window.

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

Comments The *xPos* and *yPos* parameters are given in screen coordinates for overlapped and pop-up windows and in parent-client coordinates for child windows.

An application can use the **MAKEPOINT** macro to convert the *lParam* parameter to a **POINT** data structure.

See Also **MAKEPOINT, POINT**

WM_NCACTIVATE

2.x

```
WM_NCACTIVATE
fActive = (BOOL) wParam;     /* the active/inactive flag */
```

The WM_NCACTIVATE message is sent to a window when its nonclient area needs to be changed to indicate an active or inactive state.

Parameters *fActive*

Value of *wParam*. Specifies when a title bar or icon needs to be changed to indicate an active or inactive state. The *fActive* parameter is TRUE if an active title bar or icon is to be drawn. It is FALSE for an inactive title bar or icon.

Return Value When the *fActive* parameter is FALSE, an application should return TRUE to indicate that Windows should proceed with the default processing or FALSE to prevent the caption bar or icon from being deactivated. When *fActive* is TRUE, the return value is ignored.

Comments The **DefWindowProc** function draws the title bar and title bar text in their active colors when the *fActive* parameter is TRUE and in their inactive colors when *fActive* is FALSE.

See Also **DefWindowProc**

WM_NCCALCSIZE

2.x

```
WM_NCCALCSIZE
fCalcValidRects = (BOOL) wParam;            /* valid-area flag  */
lpncsp = (NCCALCSIZE_PARAMS FAR*) lParam;   /* address of data  */
```

The WM_NCCALCSIZE message is sent when the size and position of a window's client area needs to be calculated. By processing this message, an application can control the contents of the window's client area when the size or position of the window changes.

Parameters

fCalcValidRects

Value of *wParam*. Specifies whether the application should specify which part of the client area contains valid information. Windows will copy the valid information to the specified area within the new client area. If this parameter is TRUE, the application should specify which part of the client area is valid.

lpncsp

Value of *lParam*. Points to an **NCCALCSIZE_PARAMS** data structure that contains information an application can use to calculate the new size and position of the client rectangle. The **NCCALCSIZE_PARAMS** structure has the following form:

```
typedef struct tagNCCALCSIZE_PARAMS {
    RECT           rgrc[3];
    WINDOWPOS FAR* lppos;
} NCCALCSIZE_PARAMS;
```

Regardless of the value of *fCalcValidRects*, the first rectangle in the array specified by the **rgrc** member contains the coordinates of the window. For a child window, the coordinates are relative to the parent window's client area. For top-level windows, the coordinates are screen coordinates. An application should process WM_NCCALCSIZE by modifying the **rgrc[0]** rectangle to reflect the size and position of the client area.

The **rgrc[1]** and **rgrc[2]** rectangles are valid only if *fCalcValidRects* is TRUE. In this case, the **rgrc[1]** rectangle contains the coordinates of the window before it was moved or resized. The **rgrc[2]** rectangle contains the coordinates of the window's client area before the window was moved. All coordinates are relative to the parent window or screen.

Return Value

An application should return zero if *fCalcValidRects* is FALSE.

An application can return zero or a valid combination of the following values if *fCalcValidRects* is TRUE:

Value	Meaning
WVR_ALIGNTOP, WVR_ALIGNLEFT, WVR_ALIGNBOTTOM, WVR_ALIGNRIGHT	These values, used in combination, specify that the client area of the window is to be preserved and aligned appropriately relative to the new location of the client window. For example, to align the client area to the lower-left, return WVR_ALIGNLEFT | WVR_ALIGNTOP.
WVR_HREDRAW, WVR_VREDRAW	These values, used in combination with any other values, cause the window to be completely redrawn if the client rectangle changed size horizontally or vertically. These values are similar to the CS_HREDRAW and CS_VREDRAW class styles.
WVR_REDRAW	This value causes the entire window to be redrawn. It is a combination of WVR_HREDRAW and WVR_VREDRAW.
WVR_VALIDRECTS	This value indicates that, upon return from WM_NCCALCSIZE, the **rgrc[1]** and **rgrc[2]** rectangles contain valid source and destination area rectangles, respectively. Windows combines these rectangles to calculate the area of the window that can be preserved. Windows copies any part of the window image that is within the source rectangle and clips the image to the destination rectangle. Both rectangles are in parent-relative or screen-relative coordinates. This return value allows an application to implement more elaborate client-area preservation strategies, such as centering or preserving a subset of the client area.

If *fCalcValidRects* is TRUE and an application returns zero, the old client area is preserved and is aligned with the upper-left corner of the new client area.

Comments

Redrawing of the window may occur, depending on whether CS_HREDRAW or CS_VREDRAW was specified. This is the default, backward-compatible **DefWindowProc** processing of this message (in addition to the usual client rectangle calculation described in the preceding table).

See Also

DefWindowProc, **MoveWindow**, **SetWindowPos**

WM_NCCREATE

2.x

```
WM_NCCREATE
lpcs = (CREATESTRUCT FAR*) lParam;  /* address of initialization data */
```

The WM_NCCREATE message is sent prior to the WM_CREATE message when a window is first created.

Parameters

lpcs

Value of *lParam*. Points to the **CREATESTRUCT** data structure for the window. The **CREATESTRUCT** structure has the following form:

```
typedef struct tagCREATESTRUCT {    /* cs */
    void FAR* lpCreateParams;
    HINSTANCE hInstance;
    HMENU     hMenu;
    HWND      hwndParent;
    int       cy;
    int       cx;
    int       y;
    int       x;
    LONG      style;
    LPCSTR    lpszName;
    LPCSTR    lpszClass;
    DWORD     dwExStyle;
} CREATESTRUCT;
```

Return Value

The return value is nonzero if the nonclient area is created. It is zero if an error occurs; in this case, the **CreateWindow** or **CreateWindowEx** function will return NULL.

Comments

Scroll bars are initialized (the scroll bar position and range are set), and the window text is set. Memory used internally to create and maintain the window is allocated.

See Also

CreateWindow, WM_CREATE

WM_NCDESTROY

2.x

```
WM_NCDESTROY
```

The WM_NCDESTROY message informs a window that its nonclient area is being destroyed. The **DestroyWindow** function sends the WM_NCDESTROY message to the window following the WM_DESTROY message. WM_NCDESTROY is used to free the allocated memory object associated with the window.

Parameters

This message has no parameters.

Return Value

An application should return zero if it processes this message.

Comments

This message frees any memory internally allocated for the window.

See Also

DestroyWindow, WM_NCCREATE

WM_NCHITTEST

2.x

```
WM_NCHITTEST
xPos = (int) LOWORD(lParam);    /* horizontal position of cursor */
yPos = (int) HIWORD(lParam);    /* vertical position of cursor   */
```

The WM_NCHITTEST message is sent to the window that contains the cursor or to the window that used the **SetCapture** function to capture the mouse input. It is sent every time the mouse is moved.

Parameters

xPos

Value of the low-order word of *lParam*. Specifies the x-coordinate of the cursor, in screen coordinates.

yPos

Value of the high-order word of *lParam*. Specifies the y-coordinate of the cursor, in screen coordinates.

Return Value

The return value of the **DefWindowProc** function is one of the following values indicating the position of the cursor:

Value	Meaning
HTBORDER	In the border of a window that does not have a sizing border
HTBOTTOM	In the lower horizontal border of a window
HTBOTTOMLEFT	In the lower-left corner of a window border
HTBOTTOMRIGHT	In the lower-right corner of a window border
HTCAPTION	In a title bar area
HTCLIENT	In a client area
HTERROR	On the screen background or on a dividing line between windows (same as HTNOWHERE except that the **DefWindowProc** function produces a system beep to indicate an error)
HTGROWBOX	In a size box (same as HTSIZE)
HTHSCROLL	In the horizontal scroll bar
HTLEFT	In the left border of a window
HTMAXBUTTON	In a Maximize button
HTMENU	In a menu area
HTMINBUTTON	In a Minimize button
HTNOWHERE	On the screen background or on a dividing line between windows
HTREDUCE	In a Minimize button
HTRIGHT	In the right border of a window
HTSIZE	In a size box (same as HTGROWBOX)
HTSYSMENU	In a System menu (sometimes referred to as a Control menu) or in a close button in a child window
HTTOP	In the upper horizontal border of a window
HTTOPLEFT	In the upper-left corner of a window border
HTTOPRIGHT	In the upper-right corner of a window border
HTTRANSPARENT	In a window currently covered by another window
HTVSCROLL	In the vertical scroll bar
HTZOOM	In a Maximize button

Comments

The **MAKEPOINT** macro can be used to convert the *lParam* parameter to a **POINT** structure.

Example

This example shows a portion of a subclass procedure that detects mouse messages in a static window:

```
LONG lRetVal;

case WM_NCHITTEST:
    lRetVal = DefWindowProc(hwnd, msg, wParam, lParam);
    if (lRetVal == HTTRANSPARENT) {
        .
        . /* Process mouse events in static window. */
        .
    }
    break;

default:
    CallWindowProc(lpStaticProc, hwnd, msg, wParam, lParam);
```

See Also **DefWindowProc, GetCapture**

WM_NCLBUTTONDBLCLK

2.x

```
WM_NCLBUTTONDBLCLK
nHittest = wParam;        /* hit-test code              */
xCursor = LOWORD(lParam); /* cursor horizontal position */
yCursor = HIWORD(lParam); /* cursor vertical position   */
```

The WM_NCLBUTTONDBLCLK message is sent when the user double-clicks the left mouse button while the cursor is within a nonclient area of the window.

Parameters

nHittest
Value of *wParam*. Specifies the code returned by WM_NCHITTEST. For more information, see the description of the WM_NCHITTEST message.

xCursor
Value of the low-order word of *lParam*. Specifies the horizontal position of the cursor, in screen coordinates.

yCursor
Value of the high-order word of *lParam*. Specifies the vertical position of the cursor, in screen coordinates.

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

Comments If appropriate, WM_SYSCOMMAND messages are sent.

See Also WM_NCHITTEST, WM_SYSCOMMAND

WM_NCLBUTTONDOWN

2.x

```
WM_NCLBUTTONDOWN
wHitTestCode = wParam;    /* hit-test code              */
xPos = LOWORD(lParam);    /* horizontal cursor position */
yPos = HIWORD(lParam);    /* vertical cursor position   */
```

The WM_NCLBUTTONDOWN message is sent to a window when the user presses the left mouse button while the cursor is within a nonclient area of the window.

Parameters

wHitTestCode
Value of *wParam*. Specifies the code returned by WM_NCHITTEST. For more information, see the description of the WM_NCHITTEST message.

xPos
Value of the low-order word of *lParam*. Specifies the x-coordinate of the cursor, in screen coordinates.

yPos
Value of the high-order word of *lParam*. Specifies the y-coordinate of the cursor, in screen coordinates.

Return Value

An application should return zero if it processes this message.

Comments

If appropriate, WM_SYSCOMMAND messages are sent.

See Also

WM_NCHITTEST, WM_NCLBUTTONDBLCLK, WM_NCLBUTTONUP, WM_SYSCOMMAND

WM_NCLBUTTONUP

2.x

```
WM_NCLBUTTONUP
wHitTestCode = wParam;    /* hit-test code              */
xPos = LOWORD(lParam);    /* horizontal cursor position */
yPos = HIWORD(lParam);    /* vertical cursor position   */
```

The WM_NCLBUTTONUP message is sent to a window when the user releases the left mouse button while the cursor is within a nonclient area of the window.

Parameters

wHitTestCode
Value of *wParam*. Specifies the code returned by WM_NCHITTEST. For more information, see the description of the WM_NCHITTEST message.

xPos
Value of the low-order word of *lParam*. Specifies the x-coordinate of the cursor, in screen coordinates.

yPos
Value of the high-order word of *lParam*. Specifies the y-coordinate of the cursor, in screen coordinates.

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

Comments If appropriate, WM_SYSCOMMAND messages are sent.

See Also WM_NCHITTEST, WM_NCLBUTTONDOWN, WM_NCLBUTTONUP, WM_SYSCOMMAND

WM_NCMBUTTONDBLCLK

2.x

```
WM_NCMBUTTONDBLCLK
wHitTestCode = wParam;    /* hit-test code              */
xPos = LOWORD(lParam);    /* horizontal cursor position */
yPos = HIWORD(lParam);    /* vertical cursor position   */
```

The WM_NCRBUTTONDOWN message is sent to a window when the user double-clicks the middle mouse button while the cursor is within a nonclient area of the window.

Parameters

wHitTestCode
Value of *wParam*. Specifies the code returned by WM_NCHITTEST. For more information, see the description of the WM_NCHITTEST message.

xPos
Value of the low-order word of *lParam*. Specifies the x-coordinate of the cursor, as a screen coordinate.

yPos
Value of the high-order word of *lParam*. Specifies the y-coordinate of the cursor, as a screen coordinate.

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

See Also WM_NCHITTEST, WM_NCMBUTTONDOWN, WM_NCMBUTTONUP

WM_NCMBUTTONDOWN

2.x

```
WM_NCMBUTTONDOWN
wHitTestCode = wParam;     /* hit-test code              */
xPos = LOWORD(lParam);     /* horizontal cursor position */
yPos = HIWORD(lParam);     /* vertical cursor position   */
```

The WM_NCMBUTTONDOWN message is sent to a window when the user presses the middle mouse button while the cursor is within a nonclient area of the window.

Parameters

wHitTestCode
Value of *wParam*. Specifies the code returned by WM_NCHITTEST. For more information, see the description of the WM_NCHITTEST message.

xPos
Value of the low-order word of *lParam*. Specifies the x-coordinate of the cursor, as a screen coordinate.

yPos
Value of the high-order word of *lParam*. Specifies the y-coordinate of the cursor, as a screen coordinate.

Return Value

An application should return zero if it processes this message.

See Also

WM_NCHITTEST, WM_NCMBUTTONDBLCLK, WM_NCMBUTTONUP

WM_NCMBUTTONUP

2.x

```
WM_NCMBUTTONUP
wHitTestCode = wParam;     /* hit-test code              */
xPos = LOWORD(lParam);     /* horizontal cursor position */
yPos = HIWORD(lParam);     /* vertical cursor position   */
```

The WM_NCMBUTTONUP message is sent to a window when the user releases the left mouse button while the cursor is within a nonclient area of the window.

Parameters

wHitTestCode
Value of *wParam*. Specifies the code returned by WM_NCHITTEST. For more information, see the description of the WM_NCHITTEST message.

xPos
Value of the low-order word of *lParam*. Specifies the x-coordinate of the cursor, as a screen coordinate.

yPos
Value of the high-order word of *lParam*. Specifies the y-coordinate of the cursor, as a screen coordinate.

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

See Also WM_NCHITTEST, WM_NCMBUTTONDBLCLK, WM_NCMBUTTONDOWN

WM_NCMOUSEMOVE

2.x

```
WM_NCMOUSEMOVE
wHitTestCode = wParam;    /* hit-test code              */
xPos = LOWORD(lParam);    /* horizontal cursor position */
yPos = HIWORD(lParam);    /* vertical cursor position   */
```

The WM_NCMOUSEMOVE message is sent to a window when the cursor is moved within a nonclient area of the window.

Parameters

wHitTestCode
Value of *wParam*. Specifies the code returned by WM_NCHITTEST. For more information, see the description of the WM_NCHITTEST message.

xPos
Value of the low-order word of *lParam*. Specifies the x-coordinate of the cursor, as a screen coordinate.

yPos
Value of the high-order word of *lParam*. Specifies the y-coordinate of the cursor, as a screen coordinate.

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

Comments If appropriate, WM_SYSCOMMAND messages are sent.

See Also WM_NCHITTEST, WM_SYSCOMMAND

WM_NCPAINT

2.x

```
WM_NCPAINT
```

The WM_NCPAINT message is sent to a window when its frame needs painting.

Parameters

This message has no parameters.

Return Value

An application should return zero if it processes this message.

Comments

The **DefWindowProc** function paints the window frame.

An application can intercept this message and paint its own custom window frame. The clipping region for a window is always rectangular, even if the shape of the frame is altered.

See Also

DefWindowProc

WM_NCRBUTTONDBLCLK

2.x

```
WM_NCRBUTTONDBLCLK
wHitTestCode = wParam;    /* hit-test code             */
xPos = LOWORD(lParam);    /* horizontal cursor position */
yPos = HIWORD(lParam);    /* vertical cursor position   */
```

The WM_NCRBUTTONDBLCLK message is sent to a window when the user double-clicks the right mouse button while the cursor is within a nonclient area of the window.

Parameters

wHitTestCode
Value of *wParam*. Specifies the code returned by WM_NCHITTEST. For more information, see the description of the WM_NCHITTEST message.

xPos
Value of the low-order word of *lParam*. Specifies the x-coordinate of the cursor, as a screen coordinate.

yPos
Value of the high-order word of *lParam*. Specifies the y-coordinate of the cursor, as a screen coordinate.

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

See Also WM_NCHITTEST, WM_NCRBUTTONDOWN, WM_NCRBUTTONUP

WM_NCRBUTTONDOWN

2.x

```
WM_NCRBUTTONDOWN
wHitTestCode = wParam;     /* hit-test code              */
xPos = LOWORD(lParam);     /* horizontal cursor position */
yPos = HIWORD(lParam);     /* vertical cursor position   */
```

The WM_NCRBUTTONDOWN message is sent to a window when the user presses the right mouse button while the cursor is within a nonclient area of the window.

Parameters

wHitTestCode
Value of *wParam*. Specifies the code returned by WM_NCHITTEST. For more information, see the description of the WM_NCHITTEST message.

xPos
Value of the low-order word of *lParam*. Specifies the x-coordinate of the cursor, as a screen coordinate.

yPos
Value of the high-order word of *lParam*. Specifies the y-coordinate of the cursor, as a screen coordinate.

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

See Also WM_NCHITTEST, WM_NCRBUTTONDBLCLK, WM_NCRBUTTONUP

WM_NCRBUTTONUP

2.x

```
WM_NCRBUTTONUP
wHitTestCode = wParam;     /* hit-test code              */
xPos = LOWORD(lParam);     /* horizontal cursor position */
yPos = HIWORD(lParam);     /* vertical cursor position   */
```

The WM_NCRBUTTONUP message is sent to a window when the user releases the right mouse button while the cursor is within a nonclient area of the window.

Parameters

wHitTestCode
Value of *wParam*. Specifies the code returned by WM_NCHITTEST. For more information, see the description of the WM_NCHITTEST message.

xPos
Value of the low-order word of *lParam*. Specifies the x-coordinate of the cursor, as a screen coordinate.

yPos
Value of the high-order word of *lParam*. Specifies the y-coordinate of the cursor, as a screen coordinate.

Return Value

An application should return zero if it processes this message.

See Also

WM_NCHITTEST, WM_NCRBUTTONDBLCLK, WM_NCRBUTTONDOWN

WM_NEXTDLGCTL

2.x

```
WM_NEXTDLGCTL
wCtlFocus = wParam;                /* identifies control for focus */
fHandle = (BOOL) LOWORD(lParam);   /* wParam handle flag           */
```

An application sends the WM_NEXTDLGCTL message to a dialog box procedure to set the focus to a different control in a dialog box.

Parameters

wCtlFocus
Value of *wParam*. If the *fHandle* parameter is nonzero, the *wCtlFocus* parameter is the handle of the control that receives the focus. If *fHandle* is zero, *wCtlFocus* is a flag that indicates whether the next or previous control with the WS_TABSTOP style receives the focus. If *wCtlFocus* is zero, the next control receives the focus; otherwise, the previous control with the WS_TABSTOP style receives the focus.

fHandle
Low-order word of *lParam*. Indicates how Windows uses the *wParam* parameter. If *fHandle* is nonzero, *wParam* is a handle associated with the control that receives the focus; otherwise, *wParam* is a flag that indicates whether the next or previous control with the WS_TABSTOP style receives the focus.

Return Value

An application should return zero if it processes this message.

Comments

The effect of this message differs from that of the **SetFocus** function because WM_NEXTDLGCTL modifies the border around the default button.

Do not use the **SendMessage** function to send a WM_NEXTDLGCTL message if your application will concurrently process other messages that set the control focus. In this case, use the **PostMessage** function instead.

See Also

PostMessage, **SendMessage**, **SetFocus**

WM_PAINT

2.x

```
WM_PAINT
```

The WM_PAINT message is sent when Windows or an application makes a request to repaint a portion of an application's window. The message is sent when the **UpdateWindow** or **RedrawWindow** function is called or by the **DispatchMessage** function when the application obtains a WM_PAINT message by using the **GetMessage** or **PeekMessage** function.

Parameters

This message has no parameters.

Return Value

An application should return zero if it processes this message.

Comments

The **DispatchMessage** function sends this message when there are no other messages in the application's message queue.

A window may receive internal paint messages as a result of calling the **RedrawWindow** function with the RDW_INTERNALPAINT flag set. In this case, the window may not have an update region. An application should call the **GetUpdateRect** function to determine whether the window has an update region. If **GetUpdateRect** returns zero, the application should not call the **BeginPaint** and **EndPaint** functions.

It is an application's responsibility to check for any necessary internal repainting or updating by looking at its internal data structures for each WM_PAINT message, because a WM_PAINT message may have been caused by both an invalid area and a call to the **RedrawWindow** function with the RDW_INTERNALPAINT flag set.

An internal WM_PAINT message is sent only once by Windows. After an internal WM_PAINT message is returned from the **GetMessage** or **PeekMessage** function or is sent to a window by the **UpdateWindow** function, no further WM_PAINT messages will be sent or posted until the window is invalidated or until the **RedrawWindow** function is called again with the RDW_INTERNALPAINT flag set.

See Also **BeginPaint, DispatchMessage, EndPaint, GetMessage, PeekMessage, RedrawWindow, UpdateWindow**

WM_PAINTCLIPBOARD

2.x

```
WM_PAINTCLIPBOARD
hwndViewer = (HWND) wParam;              /* handle of viewer     */
pps = (PAINTSTRUCT FAR*) LOWORD(lParam); /* points to paint data */
```

The WM_PAINTCLIPBOARD message is sent by a clipboard viewer to the clipboard owner when the owner has placed data on the clipboard in the CF_OWNERDISPLAY format and the clipboard viewer's client area needs repainting.

Parameters

hwndViewer
Value of *wParam*. Specifies a handle to the clipboard viewer window.

pps
Value of the low-order word of *lParam*. Points to a **PAINTSTRUCT** data structure that defines which part of the client area to paint. The **PAINTSTRUCT** structure has the following form:

```
typedef struct tagPAINTSTRUCT {     /* ps */
   HDC  hdc;
   BOOL fErase;
   RECT rcPaint;
   BOOL fRestore;
   BOOL fIncUpdate;
   BYTE rgbReserved[16];
} PAINTSTRUCT;
```

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

Comments To determine whether the entire client area or just a portion of it needs repainting, the clipboard owner must compare the dimensions of the drawing area given in the **rcPaint** member of the **PAINTSTRUCT** structure to the dimensions given in the most recent WM_SIZECLIPBOARD message.

An application must use the **GlobalLock** function to lock the memory that contains the **PAINTSTRUCT** data structure. The application should unlock that memory by using the **GlobalUnlock** function before it yields or returns control.

See Also **GlobalLock**, **GlobalUnlock**, WM_SIZECLIPBOARD

WM_PALETTECHANGED

3.0

```
WM_PALETTECHANGED
hwndPalChg = (HWND) wParam; /* handle of window that changed palette */
```

The WM_PALETTECHANGED message is sent to all top-level and overlapped windows after the window with the input focus has realized its logical palette, thereby changing the system palette. This message allows a window without the input focus that uses a color palette to realize its logical palette and update its client area.

Parameters *hwndPalChg*
Value of *wParam*. Specifies the handle of the window that caused the system palette to change.

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

Comments This message is sent to all top-level and overlapped windows, including the one that changed the system palette and caused this message to be sent. If any child windows use a color palette, this message must be passed on to them.

To avoid an infinite loop, a window that receives this message should not realize its palette unless it determines that *wParam* does not contain its own window handle.

Example

This example shows how an application selects and realizes its logical palette:

```
HDC hdc;
HPALETTE hpalApp, hpalT;
UINT i;

/*
 * If this application changed the palette, ignore the message.
 */

case WM_PALETTECHANGED:
    if (wParam == hwnd)
        return 0L;

/* Otherwise, fall through to WM_QUERYNEWPALETTE. */

case WM_QUERYNEWPALETTE:

    /*
     * If realizing the palette causes the palette to change,
     * redraw completely.
     */

    hdc = GetDC(hwnd);
    hpalT = SelectPalette (hdc, hpalApp, FALSE);

    i = RealizePalette(hdc); /* i == entries that changed  */

    SelectPalette (hdc, hpalT, FALSE);
    ReleaseDC(hwnd, hdc);


    /* If any palette entries changed, repaint the window. */

    if (i > 0)
        InvalidateRect(hwnd, NULL, TRUE);

    return i;
```

See Also

WM_PALETTEISCHANGING, WM_QUERYNEWPALETTE

WM_PALETTEISCHANGING

3.1

```
WM_PALETTEISCHANGING
hwndRealize = (HWND) wParam; /* handle of window to realize palette */
```

The WM_PALETTEISCHANGING message informs applications that an application is going to realize its logical palette.

Parameters

hwndRealize
Value of *wParam*. Specifies the handle of the window that is going to realize its logical palette.

Return Value

An application should return zero if it processes this message.

See Also

WM_PALETTECHANGED, WM_QUERYNEWPALETTE

WM_PARENTNOTIFY

3.0

```
WM_PARENTNOTIFY
fwEvent = wParam;          /* event flags                        */
wValue1 = LOWORD(lParam); /* child handle/cursor x-coordinate */
wValue2 = HIWORD(lParam); /* child ID/cursor y-coordinate     */
```

The WM_PARENTNOTIFY message is sent to the parent of a child window when the child window is created or destroyed or when the user clicks a mouse button while the cursor is over the child window. When the child window is being created, the system sends WM_PARENTNOTIFY just before the **CreateWindow** or **CreateWindowEx** function that creates the window returns. When the child window is being destroyed, the system sends the message before any processing to destroy the window takes place.

Parameters

fwEvent
Value of *wParam*. Specifies the event for which the parent is being notified. It can be any of the following values:

Value	Description
WM_CREATE	The child window is being created.
WM_DESTROY	The child window is being destroyed.
WM_LBUTTONDOWN	The user has placed the mouse cursor over the child window and clicked the left mouse button.

Value	Description
WM_MBUTTONDOWN	The user has placed the mouse cursor over the child window and clicked the middle mouse button.
WM_RBUTTONDOWN	The user has placed the mouse cursor over the child window and clicked the right mouse button.

wValue1
Value of the low-order word of *lParam*. If the *fwEvent* parameter is WM_CREATE or WM_DESTROY, the *wValue1* parameter specifies the handle of the child window. Otherwise, *wValue1* specifies the x-coordinate of the cursor.

wValue2
Value of the high-order word of *lParam*. If *fwEvent* is WM_CREATE or WM_DESTROY, the *wValue2* parameter specifies the identifier of the child window. Otherwise, *wValue2* specifies the y-coordinate of the cursor.

Return Value

An application should return zero if it processes this message.

Comments

This message is also sent to all ancestor windows of the child window, including the top-level window.

All child windows except those that have the WS_EX_NOPARENTNOTIFY send this message to their parent windows. By default, child windows in a dialog box have the WS_EX_NOPARENTNOTIFY style unless the **CreateWindowEx** function was called to create the child window without this style.

See Also

CreateWindow, **CreateWindowEx**, WM_CREATE, WM_DESTROY, WM_LBUTTONDOWN, WM_MBUTTONDOWN, WM_RBUTTONDOWN

WM_PASTE

2.x

```
WM_PASTE
wParam = 0;      /* not used, must be zero */
lParam = 0L;     /* not used, must be zero */
```

An application sends the WM_PASTE message to an edit control or combo box to insert the data from the clipboard into the edit control at the current cursor position. Data is inserted only if the clipboard contains data in CF_TEXT format.

Parameters This message has no parameters.

Return Value The return value is nonzero if this message is sent to an edit control or a combo box.

Example This example pastes data from the clipboard to an edit control:

```
SendDlgItemMessage(hdlg, IDD_MYEDITCONTROL, WM_PASTE, 0, 0L);
```

See Also WM_CLEAR, WM_COPY, WM_CUT

WM_POWER

3.1

```
WM_POWER
fwPowerEvt = wParam;     /* power-event notification message */
```

The WM_POWER message is sent when the system, typically a battery-powered personal computer, is about to enter the suspended mode.

Parameters *fwPowerEvt*

Value of *wParam*. Specifies a power-event notification message. This parameter may be one of the following values:

Value	Meaning
PWR_SUSPENDREQUEST	Indicates that the system is about to enter the suspended mode.
PWR_SUSPENDRESUME	Indicates that the system is resuming operation after entering the suspended mode normally—that is, the system sent a PWR_SUSPENDREQUEST notification message to the application before the system was suspended. An application should perform any necessary recovery actions.
PWR_CRITICALRESUME	Indicates that the system is resuming operation after entering the suspended mode without first sending a PWR_SUSPENDREQUEST notification message to the application. An application should perform any necessary recovery actions.

Return Value The value an application should return depends on the value of the *wParam* parameter, as follows:

Value of *wParam*	Return Value
PWR_SUSPENDREQUEST	PWR_FAIL to prevent the system from entering the suspended state; otherwise PWR_OK
PWR_SUSPENDRESUME	0
PWR_CRITICALRESUME	0

Comments

This message is sent only to an application that is running on a system that conforms to the advanced power management (APM) basic input-and-output system (BIOS) specification. The message is sent by the power-management driver to each window returned by the **EnumWindows** function.

The suspended mode is the state in which the greatest amount of power savings occurs, but all operational data and parameters are preserved. Random-access memory (RAM) contents are preserved, but many devices are likely to be turned off.

See Also

EnumWindows

WM_QUERYDRAGICON

3.0

```
WM_QUERYDRAGICON
```

The WM_QUERYDRAGICON message is sent to a minimized (iconic) window that does not have an icon defined for its class. The system sends this message whenever it needs to display an icon for the window.

Parameters

This message has no parameters.

Return Value

An application should return a doubleword value that contains a cursor or icon handle in the low-order word. The cursor or icon must be compatible with the display driver's resolution. If the application returns NULL, the system displays the default cursor. The default return value is NULL.

Comments

If an application returns the handle of an icon or cursor, the system converts the icon or cursor to black-and-white.

The application can call the **LoadCursor** or **LoadIcon** function to load a cursor or icon from the resources in its executable file and to obtain this handle.

Example

This example returns an icon handle in response to the WM_QUERYDRAGICON message. The icon is loaded from the resources in the application's executable file.

```
static HICON hIcon;

switch(msg) {
    case WM_CREATE:

        /* Load icon resource. */

        hIcon = LoadIcon(hInstance, (LPCSTR) "MyIcon");
        .
        . /* Initialize other variables. */
        .

        return 0L;

    case WM_QUERYDRAGICON:

        /* Icon is about to be dragged. Return handle to custom icon. */

        return (hIcon);

    .
    . /* Process other messages. */
    .

}
```

See Also

LoadCursor, LoadIcon

WM_QUERYENDSESSION

2.x

```
WM_QUERYENDSESSION
```

The WM_QUERYENDSESSION message is sent when the user chooses to end the Windows session, or when an application calls the **ExitWindows** function. If any application returns zero, the Windows session is not ended. Windows stops sending WM_QUERYENDSESSION messages as soon as one application returns zero and sends WM_ENDSESSION messages, with the *wParam* parameter set to FALSE, to any applications that have already returned nonzero.

Parameters This message has no parameters.

Return Value An application should return nonzero if it can conveniently terminate; otherwise, it should return zero.

Comments The **DefWindowProc** function returns nonzero when it processes this message.

See Also **DefWindowProc**, **ExitWindows**, WM_ENDSESSION

WM_QUERYNEWPALETTE

3.0

WM_QUERYNEWPALETTE

The WM_QUERYNEWPALETTE message informs an application that it is about to receive the input focus, giving the application an opportunity to realize its logical palette when it receives the focus.

Parameters This message has no parameters.

Return Value An application should return nonzero if it realizes its logical palette; otherwise, it should return zero.

Example This example shows how an application selects and realizes its logical palette:

```
HDC hdc;
HPALETTE hpalApp, hpalT;
UINT i;

/*
 * If this application changed the palette, ignore the message.
 */

case WM_PALETTECHANGED:
    if (wParam == hwnd)
        return 0L;

/* Otherwise, fall through to WM_QUERYNEWPALETTE. */
```

```
case WM_QUERYNEWPALETTE:

    /*
     * If realizing the palette causes the palette to change,
     * redraw completely.
     */

    hdc = GetDC(hwnd);
    hpalT = SelectPalette (hdc, hpalApp, FALSE);

    i = RealizePalette(hdc); /* i == entries that changed  */

    SelectPalette (hdc, hpalT, FALSE);
    ReleaseDC(hwnd, hdc);


    /* If any palette entries changed, repaint the window. */

    if (i > 0)
        InvalidateRect(hwnd, NULL, TRUE);

    return i;
```

See Also WM_PALETTECHANGED, WM_PALETTEISCHANGING

WM_QUERYOPEN

2.x

WM_QUERYOPEN

The WM_QUERYOPEN message is sent to a minimized window when the user requests that the window be restored to its preminimized size and position.

Parameters This message has no parameters.

Return Value An application that processes this message should return a nonzero value if the icon can be opened or zero to prevent the icon from opened.

Comments While processing this message, the application should not perform any action that would cause an activation or focus change (for example, creating a dialog box).

The **DefWindowProc** function returns nonzero when it processes this message.

WM_QUEUESYNC

3.1

```
WM_QUEUESYNC
```

The WM_QUEUESYNC message is sent by a computer-based training (CBT) application to separate user-input messages from other messages sent through the journal playback hook (WH_JOURNALPLAYBACK).

Parameters

This message has no parameters.

Return Value

A CBT application should return zero if it processes this message.

Comments

Whenever a CBT application uses the journal playback hook, the first and last messages rendered are WM_QUEUESYNC. This allows the CBT application to intercept and examine user-initiated messages without doing so for events that it sends.

WM_QUIT

2.x

```
WM_QUIT
wExit = wParam; /* exit code */
```

The WM_QUIT message indicates a request to terminate an application and is generated when the application calls the **PostQuitMessage** function. It causes the **GetMessage** function to return zero.

Parameters

wExit
Value of *wParam*. Specifies the exit code given in the **PostQuitMessage** function.

Return Value

This message does not have a return value, because it causes the message loop to terminate before the message is sent to the application's window procedure.

See Also

GetMessage, **PostQuitMessage**

WM_RBUTTONDBLCLK

2.x

```
WM_RBUTTONDBLCLK
fwKeys = wParam;          /* key flags                        */
xPos = LOWORD(lParam);    /* horizontal position of cursor */
yPos = HIWORD(lParam);    /* vertical position of cursor   */
```

The WM_RBUTTONDBLCLK message is sent when the user double-clicks the right mouse button.

Parameters

fwKeys
Value of *wParam*. Indicates whether various virtual keys are down. This parameter can be any combination of the following values:

Value	Description
MK_CONTROL	Set if CTRL key is down.
MK_LBUTTON	Set if left button is down.
MK_MBUTTON	Set if middle button is down.
MK_RBUTTON	Set if right button is down.
MK_SHIFT	Set if SHIFT key is down.

xPos
Value of the low-order word of *lParam*. Specifies the x-coordinate of the cursor. The coordinate is relative to the upper-left corner of the window.

yPos
Value of the high-order word of *lParam*. Specifies the y-coordinate of the cursor. The coordinate is relative to the upper-left corner of the window.

Return Value

An application should return zero if it processes this message.

Comments

Only windows that have the CS_DBLCLKS class style can receive WM_RBUTTONDBLCLK messages. Windows generates a WM_RBUTTONDBLCLK message when the user presses, releases, and again presses the right mouse button within the system's double-click time limit. Double-clicking the right mouse button actually generates four messages: a WM_RBUTTONDOWN message, a WM_RBUTTONUP message, the WM_RBUTTONDBLCLK message, and another WM_RBUTTONUP message.

See Also

WM_RBUTTONDOWN, WM_RBUTTONUP

WM_RBUTTONDOWN

2.x

```
WM_RBUTTONDOWN
fwKeys = wParam;          /* key flags                      */
xPos = LOWORD(lParam);    /* horizontal position of cursor */
yPos = HIWORD(lParam);    /* vertical position of cursor   */
```

The WM_RBUTTONDOWN message is sent when the user presses the right mouse button.

Parameters

fwKeys

Indicates whether various virtual keys are down. This parameter can be any combination of the following values:

Value	Description
MK_CONTROL	Set if CTRL key is down.
MK_LBUTTON	Set if left mouse button is down.
MK_MBUTTON	Set if middle mouse button is down.
MK_SHIFT	Set if SHIFT key is down.

xPos

Value of the low-order word of *lParam*. Specifies the x-coordinate of the cursor. The coordinate is relative to the upper-left corner of the window.

yPos

Value of the high-order word of *lParam*. Specifies the y-coordinate of the cursor. The coordinate is relative to the upper-left corner of the window.

Return Value

An application should return zero if it processes this message.

See Also

WM_RBUTTONDBLCLK, WM_RBUTTONUP

WM_RBUTTONUP

2.x

```
WM_RBUTTONUP
fwKeys = wParam;          /* key flags                      */
xPos = LOWORD(lParam);    /* horizontal position of cursor */
yPos = HIWORD(lParam);    /* vertical position of cursor   */
```

The WM_RBUTTONUP message is sent when the user releases the right mouse button.

Parameters

fwKeys

Value of *wParam*. Indicates whether various virtual keys are down. This parameter can be any combination of the following values:

Value	Description
MK_CONTROL	Set if CTRL key is down.
MK_LBUTTON	Set if left mouse button is down.
MK_MBUTTON	Set if middle mouse button is down.
MK_SHIFT	Set if SHIFT key is down.

xPos

Value of the low-order word of *lParam*. Specifies the x-coordinate of the cursor. The coordinate is relative to the upper-left corner of the window.

yPos

Value of the high-order word of *lParam*. Specifies the y-coordinate of the cursor. The coordinate is relative to the upper-left corner of the window.

Return Value

An application should return zero if it processes this message.

See Also

WM_RBUTTONDBLCLK, WM_RBUTTONDOWN

WM_RENDERALLFORMATS

2.x

```
WM_RENDERALLFORMATS
```

The WM_RENDERALLFORMATS message is sent to the clipboard owner when the owner application is being destroyed.

Parameters

This message has no parameters.

Return Value

An application should return zero if it processes this message.

Comments

The clipboard owner should render the data in all the formats it is capable of generating and pass a data handle for each format to the clipboard by calling the **SetClipboardData** function. This ensures that the clipboard contains valid data even though the application that rendered the data is destroyed. The application should call the **OpenClipboard** function before calling **SetClipboardData** and should call the **CloseClipboard** function afterward.

Example

In this example, the application sends a WM_RENDERFORMAT message to itself for each clipboard format that the application supports:

```
case WM_RENDERALLFORMATS:
    OpenClipboard(hwnd);
    SendMessage(hwnd, WM_RENDERFORMAT, CF_DIB, 0L);
    SendMessage(hwnd, WM_RENDERFORMAT, CF_BITMAP, 0L);
    CloseClipboard();
    break;
```

See Also

CloseClipboard, **OpenClipboard**, **SetClipboardData**, WM_RENDERFORMAT

WM_RENDERFORMAT

2.x

```
WM_RENDERFORMAT
uFmt = (UINT) wParam;    /* clipboard data format */
```

The WM_RENDERFORMAT message is sent to the clipboard owner when a particular format with delayed rendering needs to be rendered. The receiver should render the data in that format and pass it to the clipboard by calling the **SetClipboardData** function.

Parameters

uFmt
Specifies the data format. It can be any one of the formats described with the **SetClipboardData** function.

Return Value

An application should return zero if it processes this message.

Comments

The application should not call the **OpenClipboard** and **CloseClipboard** functions while processing this message.

Example

This example uses an application-defined function to render clipboard data. The function returns a data handle that is passed to the clipboard by the **SetClipboardData** function.

```
HANDLE hData;

case WM_RENDERFORMAT:
    if (hData = RenderFormat(wParam))
        SetClipboardData(wParam, hData);
    break;
```

See Also

CloseClipboard, **OpenClipboard**, **SetClipboardData**, WM_RENDERALLFORMATS

WM_SETCURSOR

2.x

```
WM_SETCURSOR
hwndCursor = (HWND) wParam;    /* handle of window with cursor */
nHittest = LOWORD(lParam);     /* hit-test code                */
wMouseMsg = HIWORD(lParam);    /* mouse-message number         */
```

The WM_SETCURSOR message is sent to a window if mouse input is not captured and the mouse causes cursor movement within the window.

Parameters

hwndCursor
Value of *wParam*. Specifies a handle to the window that contains the cursor.

nHittest
Value of the low-order word of *lParam*. Specifies the hit-test area code.

wMouseMsg
Value of the high-order word of *lParam*. Specifies the number of the mouse message.

Return Value

An application should return TRUE to halt further processing or FALSE to continue.

Comments

If the *nHittest* parameter is HTERROR and the *wMouseMsg* parameter is a mouse button–down message, the **MessageBeep** function is called.

The **DefWindowProc** function passes the WM_SETCURSOR message to a parent window before processing. If the parent window returns TRUE, further processing is halted. Passing the message to a window's parent window gives the parent window control over the cursor's setting in a child window. The **DefWindowProc** function also uses this message to set the cursor to a pointer if it is not in the client area or to the registered-class cursor if it is in the client area.

For a standard dialog box to set the cursor for one of its child window controls, it must force the **DefDlgProc** function to return TRUE in response to the WM_SETCURSOR message. (**DefDlgProc** provides default processing for the standard dialog box class.) When **DefDlgProc** returns TRUE, the dialog box procedure retains control over the cursor. When the dialog box procedure processes WM_SETCURSOR, it can return TRUE by using the **SetWindowLong** function and the DWL_MSGRESULT offset, as shown in the following example:

```
SetWindowLong(hwndDlg, DWL_MSGRESULT, MAKELONG(TRUE, 0));
```

See Also

DefWindowProc, **MessageBeep**, **SetWindowLong**

WM_SETFOCUS

2.x

```
WM_SETFOCUS
hwnd = (HWND) wParam;    /* handle of window losing focus */
```

The WM_SETFOCUS message is sent after a window gains the input focus.

Parameters

hwnd
Value of *wParam*. Contains the handle of the window that loses the input focus. (This parameter may be NULL.)

Return Value

An application should return zero if it processes this message.

Comments

To display a caret, an application should call the appropriate caret functions at this point.

WM_SETFONT

3.0

```
WM_SETFONT
wParam = (WPARAM) hfont;                        /* handle of the font */
lParam = (LPARAM) MAKELONG((WORD) fRedraw, 0); /* redraw flag        */
```

An application sends the WM_SETFONT message to specify the font that a control is to use when drawing text.

Parameters

hfont
Value of *wParam*. Specifies the handle of the font. If this parameter is NULL, the control will use the default system font to draw text.

fRedraw
Value of the low-order word of *lParam*. Specifies whether the control should be redrawn immediately upon setting the font. Setting the *fRedraw* parameter to TRUE causes the control to redraw itself.

Return Value

An application should return zero if it processes this message.

Comments

The WM_SETFONT message applies to all controls, not just those in dialog boxes.

The best time for the owner of a dialog box to set the font of the control is when it receives the WM_INITDIALOG message. The application should call the **DeleteObject** function to delete the font when it is no longer needed—for example, after the control is destroyed.

The size of the control is not changed as a result of receiving this message. To prevent Windows from clipping text that does not fit within the boundaries of the control, the application should correct the size of the control window before changing the font.

Before Windows creates a dialog box with the DS_SETFONT style, Windows sends the WM_SETFONT message to the dialog box window before creating the controls. An application creates a dialog box with the DS_SETFONT style by calling any of the following functions:

- **CreateDialogIndirect**
- **CreateDialogIndirectParam**
- **DialogBoxIndirect**
- **DialogBoxIndirectParam**

The **DialogBoxHeader** structure that the application passes to these functions must have the DS_SETFONT style set and must contain the **wPointSize** and **szFaceName** members that define the font the dialog box will use to draw text.

For more information about the **DialogBoxHeader** structure, see Chapter 7, "Resource Formats Within Executable Files," in the *Microsoft Windows Programmer's Reference, Volume 4.*

Example

This example changes the font used by controls in a dialog box to a font that is not bold.

```
HFONT hDlgFont;
LOGFONT lFont;

case WM_INITDIALOG:

    /* Get dialog box font and create version that is not bold. */
```

```
hDlgFont = (HFONT) NULL;
if ((hDlgFont = (HFONT) SendMessage(hdlg, WM_GETFONT, 0, 0L))) {
    if (GetObject(hDlgFont, sizeof(LOGFONT), (LPSTR) &lFont)) {
        lFont.lfWeight = FW_NORMAL;
        if (hDlgFont = CreateFontIndirect((LPLOGFONT) &lFont)) {
            SendDlgItemMessage(hdlg, ID_CTRL1, WM_SETFONT,
                hDlgFont, 0L);
            SendDlgItemMessage(hdlg, ID_CTRL2, WM_SETFONT,
                hDlgFont, 0L);
            .
            . /* Set font for remaining controls. */
            .
        }
    }
}
return TRUE;
```

See Also **CreateDialogIndirect, CreateDialogIndirectParam, DeleteObject, DialogBoxIndirect, DialogBoxIndirectParam**

WM_SETREDRAW

2.x

```
WM_SETREDRAW
wParam = (WPARAM) fRedraw;    /* state of redraw flag   */
lParam = 0L;                  /* not used, must be zero */
```

An application sends a WM_SETREDRAW message to a window to allow changes in that window to be redrawn or to prevent changes in that window from being redrawn.

Parameters

fRedraw
Value of *wParam*. Specifies the state of the redraw flag. If this parameter is nonzero, the redraw flag is set. If this parameter is zero, the flag is cleared.

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

Comments This message sets or clears the redraw flag. If the redraw flag is cleared, the contents of the specified window will not be updated after each change, and the window will not be repainted until the redraw flag is set. For example, an application that needs to add several items to a list box can clear the redraw flag, add the items, and then set the redraw flag. Finally, the application can call the **InvalidateRect** function to cause the list box to be repainted.

WM_SETTEXT

2.x

```
WM_SETTEXT
wParam = 0;                          /* not used, must be zero        */
lParam = (LPARAM) (LPCSTR) pszText; /* address of window-text string */
```

An application sends a WM_SETTEXT message to set the text of a window.

Parameters

pszText
Value of *lParam*. Points to a null-terminated string that is the window text.

Return Value

The return value is LB_ERRSPACE (for a list box) or CB_ERRSPACE (for a combo box) if insufficient space is available to set the text in the edit control. It is CB_ERR if this message is sent to a combo box without an edit control.

Comments

For an edit control, the text to be set is the contents of the edit control. For a combo box, the text is the contents of the edit-control (or static-text) portion of the combo box. For a button, the text is the button name. For other windows, the text is the window title.

This message does not change the current selection in the list box of a combo box. An application should use the CB_SELECTSTRING message to select the item in the list box that matches the text in the edit control.

See Also

WM_GETTEXT

WM_SHOWWINDOW

2.x

```
WM_SHOWWINDOW
fShow = (BOOL) wParam;              /* show/hide flag */
fnStatus = LOWORD(lParam);    /* status flag    */
```

The WM_SHOWWINDOW message is sent to a window when it is about to be hidden or shown. A window is hidden or shown when the **ShowWindow** function is called; when an overlapped window is maximized or restored; or when an overlapped or pop-up window is minimized or displayed on the screen. When an overlapped window is minimized, all pop-up windows associated with that window are hidden.

Parameters

fShow
Value of *wParam*. Specifies whether a window is being shown. It is TRUE if the window is being shown; it is FALSE if the window is being hidden.

fnStatus
Value of the low-order word of *lParam*. Specifies the status of the window being shown. The *fnStatus* parameter is zero if the message is sent because of a **ShowWindow** function call; otherwise, *fnStatus* is one of the following values:

Value	Description
SW_PARENTCLOSING	Parent window is being minimized, or a pop-up window is being hidden.
SW_PARENTOPENING	Parent window is opening (being displayed) or a pop-up window is being shown.

Return Value

An application should return zero if it processes this message.

Comments

The **DefWindowProc** function hides or shows the window as specified by the message.

The WM_SHOWWINDOW message is not sent under the following circumstances:

- When a main window is created with the WS_MAXIMIZE or WS_MINIMIZE style
- When the SW_SHOWNORMAL flag is specified in the call to the **ShowWindow** function

See Also

DefWindowProc, **ShowWindow**

WM_SIZE

2.x

```
WM_SIZE
fwSizeType = wParam;      /* sizing-type flag      */
nWidth = LOWORD(lParam);  /* width of client area  */
nHeight = HIWORD(lParam); /* height of client area */
```

The WM_SIZE message is sent to a window after its size has changed.

Parameters

fwSizeType
Value of *wParam*. Specifies the type of resizing requested. This parameter can be one of the following values:

Value	Description
SIZE_MAXIMIZED	Window has been maximized.
SIZE_MINIMIZED	Window has been minimized.
SIZE_RESTORED	Window has been resized, but neither SIZE_MINIMIZED nor SIZE_MAXIMIZED applies.
SIZE_MAXHIDE	Message is sent to all pop-up windows when some other window is maximized.
SIZE_MAXSHOW	Message is sent to all pop-up windows when some other window has been restored to its former size.

nWidth
Value of the low-order word of *lParam*. Specifies the new width of the client area.

nHeight
Value of the high-order word of *lParam*. Specifies the new height of the client area.

Return Value

An application should return zero if it processes this message.

Comments

If the **SetScrollPos** or **MoveWindow** function is called for a child window as a result of the WM_SIZE message, the *fRepaint* parameter should be nonzero to cause the window to be repainted.

See Also

MoveWindow, **SetScrollPos**

WM_SIZECLIPBOARD

2.x

```
WM_SIZECLIPBOARD
hwndViewer = (HWND) wParam;      /* handle of clipboard viewer */
hglb = (HGLOBAL) LOWORD(lParam); /* handle of global object    */
```

The WM_SIZECLIPBOARD message is sent by the clipboard viewer to the clipboard owner when the clipboard contains data with the CF_OWNERDISPLAY attribute and the size of the client area of the clipboard-viewer window has changed.

Parameters

hwndViewer
Value of *wParam*. Identifies the clipboard-application window.

hglb
Value of the low-order word of *lParam*. Identifies a global memory object that contains a **RECT** data structure. The structure specifies the area that the clipboard owner should paint. The **RECT** structure has the following form:

```
typedef struct tagRECT {    /* rc */
   int left;
   int top;
   int right;
   int bottom;
} RECT;
```

Return Value

An application should return zero if it processes this message.

Comments

A WM_SIZECLIPBOARD message is sent with a null rectangle (0,0,0,0) as the new size when the clipboard application is about to be destroyed or minimized. This permits the clipboard owner to free its display resources.

An application must use the **GlobalLock** function to lock the memory that contains the **RECT** data structure. The application should unlock that memory by using the **GlobalUnlock** function before it yields or returns control.

See Also

GlobalLock, **GlobalUnlock**, **SetClipboardData**, **SetClipboardViewer**

WM_SPOOLERSTATUS

3.0

```
WM_SPOOLERSTATUS
fwJobStatus = wParam;       /* job-status flag          */
cJobsLeft = LOWORD(lParam); /* number of jobs remaining */
```

The WM_SPOOLERSTATUS message is sent from Print Manager whenever a job is added to or removed from the Print Manager queue.

Parameters

fwJobStatus
Value of *wParam*. Specifies the SP_JOBSTATUS flag.

cJobsLeft
Value of the low-order word of *lParam*. Specifies the number of jobs remaining in the Print Manager queue.

Return Value

An application should return zero if it processes this message.

Comments

This message is for informational purposes only.

WM_SYSCHAR

2.x

```
WM_SYSCHAR
wKeyCode = wParam;       /* ASCII key code */
dwKeyData = lParam;      /* key data       */
```

The WM_SYSCHAR message is sent to the window with the input focus when a WM_SYSKEYUP and a WM_SYSKEYDOWN message are translated. It specifies the virtual-key code of the System-menu key. (The System menu is sometimes referred to as the Control menu.)

Parameters

wKeyCode
Value of *wParam*. Specifies the ASCII-character key code of a System-menu key.

dwKeyData
Value of *lParam*. Specifies the repeat count, scan code, extended key, context code, previous key state, and key-transition state, as shown in the following table:

Bit	Description
0–15	Specifies the repeat count. The value is the number of times the keystroke is repeated as a result of the user holding down the key.
16–23	Specifies the scan code. The value depends on the original equipment manufacturer (OEM).
24	Specifies whether the key is an extended key, such as a function key or a key on the numeric keypad. The value is 1 if it is an extended key; otherwise, it is 0.
25–26	Not used.
27–28	Used internally by Windows.
29	Specifies the context code. The value is 1 if the ALT key is held down while the key is pressed; otherwise, the value is 0.
30	Specifies the previous key state. The value is 1 if the key is down before the message is sent, or it is 0 if the key is up.
31	Specifies the key-transition state. The value is 1 if the key is being released, or it is 0 if the key is being pressed.

Return Value

An application should return zero if it processes this message.

Comments

When the context code is zero, the message can be passed to the **TranslateAccelerator** function, which will handle it as though it were a normal key message instead of a System-menu key message. This allows accelerator keys to be used with the active window even if the active window does not have the input focus.

For IBM Enhanced 101- and 102-key keyboards, enhanced keys are the right ALT key and the right CTRL key on the main section of the keyboard; the INS, DEL, HOME, END, PAGE UP, PAGE DOWN, and arrow keys in the clusters to the left of the numeric keypad; and the division (/) and ENTER keys on the numeric keypad. Some other keyboards may support the extended-key bit in the *lParam* parameter.

See Also **TranslateAccelerator**, WM_SYSKEYDOWN, WM_SYSKEYUP

WM_SYSCOLORCHANGE

2.x

```
WM_SYSCOLORCHANGE
```

The WM_SYSCOLORCHANGE message is sent to all top-level windows when a change is made in the system color setting.

Parameters This message has no parameters.

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

Comments Windows sends a WM_PAINT message to any window that is affected by a system color change.

Applications that have brushes that use the existing system colors should delete those brushes and re-create them by using the new system colors.

See Also **SetSysColors**, WM_PAINT

WM_SYSCOMMAND

2.x

```
WM_SYSCOMMAND
wCmdType = wParam;        /* command value                 */
xPos = LOWORD(lParam);    /* horizontal position of cursor */
yPos = HIWORD(lParam);    /* vertical position of cursor   */
```

The WM_SYSCOMMAND message is sent when the user selects a command from the System menu (sometimes referred to as the Control menu) or when the user selects the Maximize button or the Minimize button.

Parameters

wCmdType

Value of *wParam*. Specifies the type of system command requested. This parameter can be one of the following values:

Value	Meaning
SC_CLOSE	Close the window.
SC_HOTKEY	Activate the window associated with the application-specified hot key. The low-order word of *lParam* identifies the window to activate.
SC_HSCROLL	Scroll horizontally.
SC_KEYMENU	Retrieve a menu through a keystroke.
SC_MAXIMIZE (or SC_ZOOM)	Maximize the window.
SC_MINIMIZE (or SC_ICON)	Minimize the window.
SC_MOUSEMENU	Retrieve a menu through a mouse click.
SC_MOVE	Move the window.
SC_NEXTWINDOW	Move to the next window.
SC_PREVWINDOW	Move to the previous window.
SC_RESTORE	Restore window to normal position and size.
SC_SCREENSAVE	Execute the screen-saver application specified in the [boot] section of the SYSTEM.INI file.
SC_SIZE	Size the window.
SC_TASKLIST	Execute or activate the Windows Task Manager application.
SC_VSCROLL	Scroll vertically.

xPos

Value of the low-order word of *lParam*. Specifies the x-coordinate of the cursor, if a System-menu command is chosen with the mouse. Otherwise, this parameter is not used.

yPos

Value of the high-order word of *lParam*. Specifies the y-coordinate of the cursor, if a System-menu command is chosen with the mouse. Otherwise, this parameter is not used.

Return Value

An application should return zero if it processes this message.

Comments

The **DefWindowProc** function carries out the System-menu request for the predefined actions specified in the preceding table.

In WM_SYSCOMMAND messages, the four low-order bits of the *wCmdType* parameter are used internally by Windows. When an application tests the value of *wCmdType*, it must combine the value 0xFFF0 with the *wCmdType* value by using the bitwise AND operator to obtain the correct result.

The menu items in a System menu can be modified by using the **GetSystemMenu**, **AppendMenu**, **InsertMenu**, and **ModifyMenu** functions. Applications that modify the System menu must process WM_SYSCOMMAND messages. Any WM_SYSCOMMAND messages not handled by the application must be passed to the **DefWindowProc** function. Any command values added by an application must be processed by the application and cannot be passed to **DefWindowProc**.

An application can carry out any system command at any time by passing a WM_SYSCOMMAND message to the **DefWindowProc** function.

Accelerator keystrokes that are defined to select items from the System menu are translated into WM_SYSCOMMAND messages; all other accelerator key strokes are translated into WM_COMMAND messages.

See Also **AppendMenu**, **DefWindowProc**, **GetSystemMenu**, **InsertMenu**, **ModifyMenu**, WM_COMMAND

WM_SYSDEADCHAR

2.x

```
WM_SYSDEADCHAR
wDeadKey = wParam;              /* dead-key character */
cRepeat = (int) LOWORD(lParam); /* repeat count       */
cAutoRepeat = HIWORD(lParam);   /* autorepeat count   */
```

The WM_SYSDEADCHAR message is sent to the window with the input focus when WM_SYSKEYUP and WM_SYSKEYDOWN messages are translated. It specifies the character value of a dead key.

Parameters

wDeadKey
Value of *wParam*. Specifies the dead-key character value.

cRepeat
Value of the low-order word of *lParam*. Specifies the repeat count.

cAutoRepeat
Value of the high-order word of *lParam*. Specifies the auto-repeat count.

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

See Also WM_SYSKEYDOWN, WM_SYSKEYUP

WM_SYSKEYDOWN

2.x

```
WM_SYSKEYDOWN
wVkey = wParam;          /* virtual-key code */
dwKeyData = lParam;      /* key data         */
```

The WM_SYSKEYDOWN message is sent to the window with the input focus when the user holds down the ALT key and then presses another key. If no window currently has the input focus, the WM_SYSKEYDOWN message is sent to the active window. The window that receives the message can distinguish between these two contexts by checking the context code in the *dwKeyData* parameter.

Parameters

wVkey

Value of *wParam*. Specifies the virtual-key code of the key being pressed.

dwKeyData

Value of *lParam*. Specifies the repeat count, scan code, extended key, context code, previous key state, and key-transition state, as shown in the following table:

Bit	Description
0–15	Specifies the repeat count. The value is the number of times the keystroke is repeated as a result of the user holding down the key.
16–23	Specifies the scan code. The value depends on the original equipment manufacturer (OEM).
24	Specifies whether the key is an extended key, such as a function key or a key on the numeric keypad. The value is 1 if it is an extended key; otherwise, it is 0.
25–26	Not used.
27–28	Used internally by Windows.
29	Specifies the context code. The value is 1 if the ALT key is held down while the key is pressed; otherwise, the value is 0.
30	Specifies the previous key state. The value is 1 if the key is down before the message is sent, or it is 0 if the key is up.
31	Specifies the key-transition state. The value is 1 if the key is being released, or it is 0 if the key is being pressed.

For WM_SYSKEYDOWN messages, the value of bit 29 (context code) is 1 if the ALT key is down while the key is pressed; it is 0 if the message is sent to the active window because no window has the input focus. The value of bit 31 (key-transition state) is 0.

Return Value

An application should return zero if it processes this message.

Comments

When the context code is zero, the message can be passed to the **Translate-Accelerator** function, which will handle it as though it were a normal key message instead of a system-key message. This allows accelerator keys to be used with the active window even if the active window does not have the input focus.

Because of the autorepeat feature, more than one WM_SYSKEYDOWN message may occur before a WM_SYSKEYUP message is sent. The previous key state (bit 30) can be used to determine whether the WM_SYSKEYDOWN message indicates the first down transition or a repeated down transition.

For IBM Enhanced 101- and 102-key keyboards, enhanced keys are the right ALT key and the right CTRL key on the main section of the keyboard; the INS, DEL, HOME, END, PAGE UP, PAGE DOWN, and arrow keys in the clusters to the left of the numeric keypad; and the division (/) and ENTER keys on the numeric keypad. Some other keyboards may support the extended-key bit in the *lParam* parameter.

See Also

TranslateAccelerator, WM_SYSKEYUP

WM_SYSKEYUP

2.x

```
WM_SYSKEYUP
wVkey = wParam;          /* virtual-key code */
dwKeyData = lParam;      /* key data         */
```

The WM_SYSKEYUP message is sent to the window with the input focus when the user releases a key that was pressed while the ALT key was held down. If no window currently has the input focus, the WM_SYSKEYUP message is sent to the active window. The window that receives the message can distinguish between these two contexts by checking the context code in the *lParam* parameter.

Parameters

wVkey

Value of *wParam*. Specifies the virtual-key code of the key being pressed.

dwKeyData

Value of *lParam*. Specifies the repeat count, scan code, extended key, context code, previous key state, and key-transition state, as shown in the following table:

Bit	Description
0–15	Specifies the repeat count. The value is the number of times the keystroke is repeated as a result of the user holding down the key.
16–23	Specifies the scan code. The value depends on the original equipment manufacturer (OEM).

Bit	Description
24	Specifies whether the key is an extended key, such as a function key or a key on the numeric keypad. The value is 1 if it is an extended key; otherwise, it is 0.
25–26	Not used.
27–28	Used internally by Windows.
29	Specifies the context code. The value is 1 if the ALT key is held down while the key is pressed; otherwise, the value is 0.
30	Specifies the previous key state. The value is 1 if the key is down before the message is sent, or it is 0 if the key is up.
31	Specifies the key-transition state. The value is 1 if the key is being released, or it is 0 if the key is being pressed.

For WM_SYSKEYUP messages, the value of bit 29 (context code) is 1 if the ALT key is down while the key is pressed; it is 0 if the message is sent to the active window because no window has the input focus. The value of bit 31 (key-transition state) is 1.

Return Value

An application should return zero if it processes this message.

Comments

When the context code is zero, the message can be passed to the **TranslateAccelerator** function, which will handle it as though it were a normal key message instead of a system-key message. This allows accelerator keys to be used with the active window even if the active window does not have the input focus.

For IBM Enhanced 101- and 102-key keyboards, enhanced keys are the right ALT key and the right CTRL key on the main section of the keyboard; the INS, DEL, HOME, END, PAGE UP, PAGE DOWN, and arrow keys in the clusters to the left of the numeric keypad; and the division (/) and ENTER keys on the numeric keypad. Some other keyboards may support the extended-key bit in the *lParam* parameter.

For non-U.S. Enhanced 102-key keyboards, the right ALT key is handled as the CTRL+ALT key combination. The following list shows the messages that result when the user presses and releases this key, in the sequence they occur:

1	WM_KEYDOWN	VK_CONTROL
2	WM_KEYDOWN	VK_MENU
3	WM_KEYUP	VK_CONTROL
4	WM_SYSKEYUP	VK_MENU

See Also

TranslateAccelerator, WM_SYSKEYDOWN

WM_SYSTEMERROR

3.1

```
WM_SYSTEMERROR
wErrSpec = wParam;  /* specifies when error occurred */
```

The WM_SYSTEMERROR message is sent when the Windows kernel encounters an error but cannot display the system-error message box.

Parameters

wErrSpec
Value of *wParam*. Specifies when the error occurred. Currently, the only valid value is 1, indicating that the error occurred when a task or library was terminating.

Return Value

An application should return zero if it processes this message.

Comments

A shell application should process this message, displaying a message box that indicates an error has occurred.

WM_TIMECHANGE

2.x

```
WM_TIMECHANGE
wParam = 0;      /* not used, must be zero */
lParam = 0L;     /* not used, must be zero */
```

An application sends the WM_TIMECHANGE message to all top-level windows after changing the system time.

Parameters

This message has no parameters.

Return Value

An application should return zero if it processes this message.

Comments

Any application that changes the system time should send this message to all top-level windows. To send the WM_TIMECHANGE message to all top-level windows, an application can use the **SendMessage** function with the *hwnd* parameter set to HWND_BROADCAST.

See Also

SendMessage

WM_TIMER

2.x

```
WM_TIMER
wTimerID = wParam;                  /* timer identifier              */
tmprc = (TIMERPROC FAR*) lParam;    /* address of timer callback     */
```

The WM_TIMER message is posted to the installing application's message queue or sent to the appropriate **TimerProc** callback function after each interval specified in the **SetTimer** function used to install a timer.

Parameters

wTimerID
Value of *wParam*. Specifies the identifier of the timer.

tmprc
Value of *lParam*. Points to a callback function that was passed to the **SetTimer** function when the timer was installed. If the *tmprc* parameter is not NULL, the system passes the WM_TIMER message to the specified callback function rather than posting the message to the application's message queue.

Return Value

An application should return zero if it processes this message.

Comments

The **DispatchMessage** function sends this message when no other messages are in the application's message queue.

Example

This example uses the WM_TIMER message to create a blinking effect for a line of text:

```
DWORD dwXYVal;
WORD wXVal, wYVal;
char szMessage[16];

case WM_TIMER:
    hdc = GetDC(hwnd);
    dwXYVal = GetTextExtent(hdc, (LPCSTR) szMessage,
        lstrlen(szMessage));
    wXVal = LOWORD(dwXYVal);
    wYVal = HIWORD(dwXYVal);
    PatBlt(hdc, 10, 10, (int) wXVal, (int) wYVal, PATINVERT);
    ReleaseDC(hwnd, hdc);
    ValidateRect(hwnd, NULL);
    break;
```

See Also

SetTimer, **TimerProc**

WM_UNDO

2.x

WM_UNDO

An application sends the WM_UNDO message to an edit control to undo the last operation. When this message is sent to an edit control, the previously deleted text is restored or the previously added text is deleted.

Parameters

This message has no parameters.

Return Value

The return value is nonzero if the operation is successful, or it is zero if an error occurs.

See Also

WM_CLEAR, WM_COPY, WM_CUT, WM_PASTE

WM_USER

2.x

WM_USER

WM_USER is a constant used by applications to help define private messages.

Comments

The WM_USER constant is used to distinguish between message values that are reserved for use by Windows and values that can be used by an application to send messages within a private window class. There are four ranges of message numbers:

Range	Meaning
0 through WM_USER – 1	Messages reserved for use by Windows.
WM_USER through 0x7FFF	Integer messages for use by private window classes.
0x8000 through 0xBFFF	Messages reserved for use by Windows.
0xC000 through 0xFFFF	String messages for use by applications.

Message numbers in the first range (0 through WM_USER – 1) are defined by Windows. Values in this range that are not explicitly defined are reserved for future use by Windows. This chapter describes messages in this range.

Message numbers in the second range (WM_USER through 0x7FFF) can be defined and used by an application to send messages within a private window class. These values cannot be used to define messages that are meaningful throughout an application, because some predefined window classes already define values in this range. For example, such predefined control classes as BUTTON, EDIT,

LISTBOX, and COMBOBOX may use these values. Messages in this range should not be sent to other applications unless the applications have been designed to exchange messages and to attach the same meaning to the message numbers.

Message numbers in the third range (0x8000 through 0xBFFF) are reserved for future use by Windows.

Message numbers in the fourth range (0xC000 through 0xFFFF) are defined at run time when an application calls the **RegisterWindowMessage** function to obtain a message number for a string. All applications that register the same string can use the associated message number for exchanging messages. The actual message number, however, is not a constant and cannot be assumed to be the same in different Windows sessions.

See Also **RegisterWindowMessage**

WM_VKEYTOITEM

3.0

```
WM_VKEYTOITEM
wVkey = wParam;                    /* virtual-key code       */
hwndLB = (HWND) LOWORD(lParam);    /* handle of the list box */
nCaretPos = HIWORD(lParam);        /* caret position         */
```

The WM_VKEYTOITEM message is sent by a list box with the LBS_WANTKEYBOARDINPUT style to its owner in response to a WM_KEYDOWN message.

Parameters

wVkey
Value of *wParam*. Specifies the virtual-key code of the key that the user pressed.

hwndLB
Value of the low-order word of *lParam*. Identifies the list box.

nCaretPos
Value of the high-order word of *lParam*. Specifies the current position of the caret.

Return Value

The return value specifies the action that the application performed in response to the message. A return value of –2 indicates that the application handled all aspects of selecting the item and requires no further action by the list box. A return value of –1 indicates that the list box should perform the default action in response to the keystroke. A return value of 0 or greater specifies the zero-based index of an

item in the list box and indicates that the list box should perform the default action for the keystroke on the given item.

Comments

Only list boxes that have the LBS_HASSTRINGS style can receive this message.

See Also

WM_CHARTOITEM, WM_KEYDOWN

WM_VSCROLL

2.x

```
WM_VSCROLL
wScrollCode = wParam;      /* scroll bar code             */
nPos = LOWORD(lParam);     /* current scroll box position */
hwndCtl = (HWND) HIWORD(lParam); /* handle of the control */
```

The WM_VSCROLL message is sent to a window when the user clicks the window's vertical scroll bar.

Parameters

wScrollCode

Value of *wParam*. Specifies a scroll bar code that indicates the user's scrolling request. This parameter can be one of the following values:

Value	Description
SB_BOTTOM	Scroll to bottom.
SB_ENDSCROLL	End scroll.
SB_LINEDOWN	Scroll one line down.
SB_LINEUP	Scroll one line up.
SB_PAGEDOWN	Scroll one page down.
SB_PAGEUP	Scroll one page up.
SB_THUMBPOSITION	Scroll to absolute position. The current position is specified by the *nPos* parameter.
SB_THUMBTRACK	Drag scroll box (thumb) to specified position. The current position is specified by the *nPos* parameter.
SB_TOP	Scroll to top.

nPos

Value of the low-order word of *lParam*. Specifies the current position of the scroll box if *wScrollCode* is SB_THUMBPOSITION or SB_THUMBTRACK; otherwise, this parameter is not used.

hwndCtl
Value of the high-order word of *lParam*. Identifies the control if WM_VSCROLL is sent by a scroll bar. If WM_VSCROLL is sent as a result of the user clicking a pop-up window's scroll bar, the high-order word is not used.

Return Value

An application should return zero if it processes this message.

Comments

The SB_THUMBTRACK message typically is used by applications that give some feedback while the scroll box is being dragged.

If an application scrolls the contents of the window, it must also reset the position of the scroll box by using the **SetScrollPos** function.

See Also

SetScrollPos, WM_HSCROLL

WM_VSCROLLCLIPBOARD

2.x

```
WM_VSCROLLCLIPBOARD
hwndViewer = (HWND) wParam;      /* handle of clipboard viewer */
wScrollCode = LOWORD(lParam);    /* scroll bar code            */
wThumbPos = HIWORD(lParam);      /* scroll box position        */
```

The WM_HSCROLLCLIPBOARD message is sent by the clipboard viewer to the clipboard owner when the clipboard data has the CF_OWNERDISPLAY format and there is an event in the clipboard viewer's vertical scroll bar. The owner should scroll the clipboard image, invalidate the appropriate section, and update the scroll bar values.

Parameters

hwndViewer
Value of *wParam*. Specifies a handle to a clipboard-viewer window.

wScrollCode
Value of the low-order word of *lParam*. Specifies one of the following scroll bar values:

Value	Description
SB_BOTTOM	Scroll to lower right.
SB_ENDSCROLL	End scroll.
SB_LINEDOWN	Scroll one line down.
SB_LINEUP	Scroll one line up.
SB_PAGEDOWN	Scroll one page down.
SB_PAGEUP	Scroll one page up.

Value	Description
SB_THUMBPOSITION	Scroll to absolute position.
SB_TOP	Scroll to upper left.

wThumbPos
Value of the high-order word of *lParam*. Specifies the scroll box position if the scroll bar code is SB_THUMBPOSITION; otherwise, the high-order word is not used.

Return Value

An application should return zero if it processes this message.

Comments

The clipboard owner should use the **InvalidateRect** function or repaint the window as needed. The scroll bar position should also be reset.

See Also

InvalidateRect, WM_HSCROLLCLIPBOARD

WM_WINDOWPOSCHANGED

3.1

```
WM_WINDOWPOSCHANGED
pwp = (const WINDOWPOS FAR*) lParam;    /* structure address          */
```

The WM_WINDOWPOSCHANGED message is sent to a window whose size, position, or z-order has changed as a result of a call to **SetWindowPos** or another window-management function.

Parameters

pwp
Value of *lParam*. Points to a **WINDOWPOS** data structure that contains information about the window's new size and position. The **WINDOWPOS** structure has the following form:

```
typedef struct tagWINDOWPOS { /* wp */
    HWND    hwnd;
    HWND    hwndInsertAfter;
    int     x;
    int     y;
    int     cx;
    int     cy;
    UINT    flags;
} WINDOWPOS;
```

Return Value

An application should return zero if it processes this message.

Comments

The **DefWindowProc** function, when it processes the WM_WINDOWPOSCHANGED message, sends the WM_SIZE and WM_MOVE messages to the window. These messages are not sent if an application handles the WM_WINDOWPOSCHANGED message without calling **DefWindowProc**. It is more efficient to perform any move or size change processing during the WM_WINDOWPOSCHANGED message without calling **DefWindowProc**.

See Also

WM_MOVE, WM_SIZE, WM_WINDOWPOSCHANGING

WM_WINDOWPOSCHANGING

3.1

```
WM_WINDOWPOSCHANGING
pwp = (WINDOWPOS FAR*) lParam;  /* address of WINDOWPOS structure */
```

The WM_WINDOWPOSCHANGING message is sent to a window whose size, position, or z-order is about to change as a result of a call to **SetWindowPos** or another window-management function.

Parameters

pwp
Value of *lParam*. Points to a **WINDOWPOS** data structure that contains information about the window's new size and position. The **WINDOWPOS** structure has the following form:

```
typedef struct tagWINDOWPOS { /* wp */
    HWND    hwnd;
    HWND    hwndInsertAfter;
    int     x;
    int     y;
    int     cx;
    int     cy;
    UINT    flags;
} WINDOWPOS;
```

Return Value

An application should return zero if it processes this message.

Comments

During this message, modifying any of the values in the **WINDOWPOS** structure affects the new size, position, or z-order. An application can prevent changes to the window by setting or clearing the appropriate bits in the **flags** member of the **WINDOWPOS** structure.

For a window with the WS_OVERLAPPED or WS_THICKFRAME style, the **DefWindowProc** function handles a WM_WINDOWPOSCHANGING message by sending a WM_GETMINMAXINFO message to the window. This is

done to validate the new size and position of the window and to enforce the CS_BYTEALIGNCLIENT and CS_BYTEALIGN client styles. An application can override this by not passing the WM_WINDOWPOSCHANGING message to the **DefWindowProc** function.

See Also WM_WINDOWPOSCHANGED

WM_WININICHANGE

2.x

```
WM_WININICHANGE
wParam = 0;                              /* not used, must be zero */
lParam = (LPARAM) (LPCSTR) pszSection; /* address of string     */
```

An application sends the WM_WININICHANGE message to all top-level windows after making a change to the Windows initialization file, WIN.INI. The **SystemParametersInfo** function sends the WM_WININICHANGE message after an application uses the function to change a setting in the WIN.INI file.

Parameters

pszSection
Value of *lParam*. Points to a string that specifies the name of the section that has changed (the string does not include the square brackets that enclose the section name).

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

Comments

To send the WM_WININICHANGE message to all top-level windows, an application can use the **SendMessage** function with the *hwnd* parameter set to HWND_BROADCAST.

If an application changes many different sections in WIN.INI at the same time, the application should send the WM_WININICHANGE message once with the *pszSection* parameter set to NULL. Otherwise, an application should send a separate WM_WININICHANGE message for each change it makes to WIN.INI.

If an application receives a WM_WININICHANGE message with the *pszSection* parameter set to NULL, the application should check all sections in WIN.INI that affect the application.

See Also **SendMessage, SystemParametersInfo**

2.2 Notification Messages

Notification messages notify a control's parent window of actions that occur within the control. Controls use the WM_COMMAND message to notify the parent window of actions that occur within the control. The *wParam* parameter of the WM_COMMAND message contains the control identifier; the low-order word of the *lParam* parameter contains the handle of the control; and the high-order word of *lParam* contains the control notification message.

This section lists notification messages in alphabetic order.

BN_CLICKED

2.x

BN_CLICKED

The BN_CLICKED notification message is sent when the user clicks a button. This notification is provided for compatibility with applications written prior to Windows version 3.0. New applications should use the BS_OWNERDRAW button style and the **DRAWITEMSTRUCT** structure for this task.

See Also **DRAWITEMSTRUCT, WM_DRAWITEM**

BN_DISABLE

2.x

BN_DISABLE

The BN_DISABLE notification message is sent when a button is disabled. This notification is provided for compatibility with applications written prior to Windows version 3.0. New applications should use the BS_OWNERDRAW button style and the **DRAWITEMSTRUCT** structure for this task.

See Also **DRAWITEMSTRUCT, WM_DRAWITEM**

BN_DOUBLECLICKED

2.x

BN_DOUBLECLICKED

The BN_DOUBLECLICKED notification message is sent when the user double clicks a button. This notification is provided for compatibility with applications written prior to Windows version 3.0. New applications should use the BS_OWNERDRAW button style and the **DRAWITEMSTRUCT** structure for this task.

See Also **DRAWITEMSTRUCT, WM_DRAWITEM**

BN_HILITE

2.x

BN_HILITE

The BN_HILITE notification message is sent when the user highlights a button. This notification is provided for compatibility with applications written prior to Windows version 3.0. New applications should use the BS_OWNERDRAW button style and the **DRAWITEMSTRUCT** structure for this task.

See Also **DRAWITEMSTRUCT, WM_DRAWITEM**

BN_PAINT

2.x

BN_PAINT

The BN_PAINT notification message is sent when a button should be painted. This notification is provided for compatibility with applications written prior to Windows version 3.0. New applications should use the BS_OWNERDRAW button style and the **DRAWITEMSTRUCT** structure for this task.

See Also **DRAWITEMSTRUCT, WM_DRAWITEM**

BN_UNHILITE

2.x

BN_UNHILITE

The BN_UNHILITE notification message is sent when the highlight should be removed from a button. This notification is provided for compatibility with applications written prior to Windows version 3.0. New applications should use the BS_OWNERDRAW button style and the **DRAWITEMSTRUCT** structure for this task.

See Also **DRAWITEMSTRUCT, WM_DRAWITEM**

CBN_CLOSEUP

3.1

The CBN_CLOSEUP notification message is sent when the list box of a combo box is hidden. The control's parent window receives this notification message through a WM_COMMAND message.

Parameters

wParam
Specifies the identifier of the combo box.

lParam
Specifies the handle of the combo box in the low-order word, and specifies the CBN_CLOSEUP notification message in the high-order word.

Comments

This notification message is not sent to a combo box that has the CBS_SIMPLE style.

The order in which notifications will be sent cannot be predicted. In particular, a CBN_SELCHANGE notification may occur either before or after a CBN_CLOSEUP notification.

See Also CBN_DROPDOWN, CBN_SELCHANGE, WM_COMMAND

CBN_DBLCLK

3.0

The CBN_DBLCLK notification message is sent when the user double-clicks a string in the list box of a combo box. The control's parent window receives this notification message through a WM_COMMAND message.

Parameters

wParam
Specifies the identifier of the combo box.

lParam
Specifies the handle of the combo box in the low-order word and the CBN_DBLCLK notification message in the high-order word.

Comments

This notification message can occur only for a combo box with the CBS_SIMPLE style. For a combo box with the CBS_DROPDOWN or CBS_DROPDOWNLIST style, a double-click cannot occur because a single click hides the list box.

See Also

CBN_SELCHANGE, WM_COMMAND

CBN_DROPDOWN

3.0

The CBN_DROPDOWN notification message is sent when the list box of a combo box is about to be dropped down (made visible). The parent window of the combo box receives this notification message through a WM_COMMAND message.

Parameters

wParam
Specifies the identifier of the combo box.

lParam
Specifies the handle of the combo box in the low-order word, and specifies the CBN_DROPDOWN notification message in the high-order word.

Comments

This notification message can occur only for a combo box with the CBS_DROPDOWN or CBS_DROPDOWNLIST style.

See Also

CBN_CLOSEUP, WM_COMMAND

CBN_EDITCHANGE

3.0

The CBN_EDITCHANGE notification message is sent after the user has taken an action that may have altered the text in the edit-control portion of a combo box. Unlike the CBN_EDITUPDATE notification message, this notification message is sent after Windows updates the screen. The parent window of the combo box receives this notification message through a WM_COMMAND message.

Parameters

wParam
Specifies the identifier of the combo box.

lParam
Specifies the handle of the combo box in the low-order word, and specifies the CBN_EDITCHANGE notification message in the high-order word.

Comments

This message does not occur if the combo box has the CBS_DROPDOWNLIST style.

See Also

CBN_EDITUPDATE, WM_COMMAND

CBN_EDITUPDATE

3.0

The CBN_EDITUPDATE notification message is sent when the edit-control portion of a combo box is about to display altered text. This notification is sent after the control has formatted the text, but before it displays the text. The parent window of the combo box receives this notification message through a WM_COMMAND message.

Parameters

wParam
Specifies the identifier of the combo box.

lParam
Specifies the handle of the combo box in the low-order word, and specifies the CBN_EDITUPDATE notification message in the high-order word.

Comments

This message does not occur if the combo box has the CBS_DROPDOWNLIST style.

See Also

CBN_EDITCHANGE, WM_COMMAND

CBN_ERRSPACE

3.0

The CBN_ERRSPACE notification message is sent when a combo box cannot allocate enough memory to meet a specific request. The parent window of the combo box receives this notification message through a WM_COMMAND message.

Parameters

wParam
Specifies the identifier of the combo box.

lParam
Specifies the handle of the combo box in the low-order word, and specifies the CBN_ERRSPACE notification message in the high-order word.

See Also WM_COMMAND

CBN_KILLFOCUS

3.0

The CBN_KILLFOCUS notification message is sent when a combo box loses the input focus. The parent window of the combo box receives this notification message through a WM_COMMAND message.

Parameters

wParam
Specifies the identifier of the combo box.

lParam
Specifies the handle of the combo box in the low-order word, and specifies the CBN_KILLFOCUS notification message in the high-order word.

See Also CBN_SETFOCUS, WM_COMMAND

CBN_SELCHANGE

3.0

The CBN_SELCHANGE notification message is sent when the selection in the list box of a combo box is about to be changed as a result of the user either clicking in the list box or changing the selection by using the arrow keys. The parent window of the combo box receives this code through a WM_COMMAND message.

Parameters

wParam
Specifies the identifier of the combo box.

lParam
Specifies the handle of the combo box in the low-order word, and specifies the CBN_SELCHANGE notification message in the high-order word.

See Also CBN_DBLCLK, CB_SETCURSEL, WM_COMMAND

CBN_SELENDCANCEL

3.1

The CBN_SELENDCANCEL notification message is sent when the user clicks an item and then clicks another window or control to hide the list box of a combo box. This notification message is sent before the CBN_CLOSEUP notification message to indicate that the user's selection should be ignored.

Parameters

wParam
Specifies the identifier of the combo box.

lParam
Specifies the handle of the combo box in the low-order word, and specifies the CBN_SELENDCANCEL notification message in the high-order word.

Comments The CBN_SELENDCANCEL or CBN_SELENDOK notification message is sent even if the CBN_CLOSEUP notification message is not sent (as in the case of a combo box with the CBS_SIMPLE style).

See Also CBN_SELENDOK, WM_COMMAND

CBN_SELENDOK

3.1

The CBN_SELENDOK notification message is sent when the user selects an item and then either presses the ENTER key or clicks the DOWN ARROW key to hide the list box of a combo box. This notification message is sent before the CBN_CLOSEUP notification message to indicate that the user's selection should be considered valid.

Parameters

wParam
Specifies the identifier of the combo box.

lParam
Specifies the handle of the combo box in the low-order word, and specifies the CBN_SELENDOK notification message in the high-order word.

Comments

The CBN_SELENDOK or CBN_SELENDCANCEL notification message is sent even if the CBN_CLOSEUP notification message is not sent (as in the case of a combo box with the CBS_SIMPLE style).

See Also

CBN_SELENDCANCEL, WM_COMMAND

CBN_SETFOCUS

3.0

The CBN_SETFOCUS notification message is sent when a combo box receives the input focus. The parent window of the combo box receives this notification message through a WM_COMMAND message.

Parameters

wParam
Specifies the identifier of the combo box.

lParam
Specifies the handle of the combo box in the low-order word, and specifies the CBN_SETFOCUS notification message in the high-order word.

See Also

CBN_KILLFOCUS, WM_COMMAND

EN_CHANGE

2.x

The EN_CHANGE notification message is sent when the user has taken an action that may have altered text in an edit control. Unlike the EN_UPDATE notification message, this notification message is sent after Windows updates the display. The control's parent window receives this notification message through a WM_COMMAND message.

Parameters

wParam
Specifies the identifier of the edit control.

lParam
Specifies the handle of the edit control in the low-order word, and specifies the EN_CHANGE notification message in the high-order word.

See Also EN_UPDATE, WM_COMMAND

EN_ERRSPACE

2.x

The EN_ERRSPACE notification message is sent when an edit control cannot allocate enough memory to meet a specific request. The control's parent window receives this notification message through a WM_COMMAND message.

Parameters

wParam
Specifies the identifier of the edit control.

lParam
Specifies the handle of the edit control in the low-order word, and specifies the EN_ERRSPACE notification message in the high-order word.

See Also WM_COMMAND

EN_HSCROLL

2.x

EN_HSCROLL

The EN_HSCROLL notification message is sent when the user clicks an edit control's horizontal scroll bar. The control's parent window receives this notification message through a WM_COMMAND message. The parent window is notified before the screen is updated.

Parameters

wParam
Specifies the identifier of the edit control.

lParam
Specifies the handle of the edit control in the low-order word, and specifies the EN_HSCROLL notification message in the high-order word.

See Also EN_VSCROLL, WM_COMMAND

EN_KILLFOCUS

2.x

The EN_KILLFOCUS notification message is sent when an edit control loses the input focus. The control's parent window receives this notification message through a WM_COMMAND message.

Parameters

wParam
Specifies the identifier of the edit control.

lParam
Specifies the handle of the edit control in the low-order word, and specifies the EN_KILLFOCUS notification message in the high-order word.

See Also

EN_SETFOCUS, WM_COMMAND

EN_MAXTEXT

3.0

The EN_MAXTEXT notification message is sent when the current insertion has exceeded the specified number of characters for the edit control. The insertion has been truncated.

This message is also sent when an edit control does not have the ES_AUTOHSCROLL style and the number of characters to be inserted would exceed the width of the edit control.

This message is also sent when an edit control does not have the ES_AUTOVSCROLL style and the total number of lines resulting from a text insertion would exceed the height of the edit control.

The control's parent window receives this notification message through a WM_COMMAND message.

Parameters

wParam
Specifies the identifier of the edit control.

lParam
Specifies the handle of the edit control in the low-order word, and specifies the EN_MAXTEXT notification message in the high-order word.

See Also

EM_LIMITTEXT, WM_COMMAND

EN_SETFOCUS

2.x

EN_SETFOCUS

The EN_SETFOCUS notification message is sent when an edit control receives the input focus. The control's parent window receives this notification message through a WM_COMMAND message.

Parameters

wParam
Specifies the identifier of the edit control.

lParam
Specifies the handle of the edit control in the low-order word, and specifies the EN_SETFOCUS notification message in the high-order word.

See Also

EN_KILLFOCUS, WM_COMMAND

EN_UPDATE

2.x

EN_UPDATE

The EN_UPDATE notification message is sent when an edit control is about to screen altered text. This notification is sent after the control has formatted the text but before it screens the text. This makes it possible to alter the window size, if necessary. The control's parent window receives this notification message through a WM_COMMAND message.

Parameters

wParam
Specifies the identifier of the edit control.

lParam
Specifies the handle of the edit control in the low-order word, and specifies the EN_UPDATE notification message in the high-order word.

See Also

EN_CHANGE, WM_COMMAND

EN_VSCROLL

2.x

EN_VSCROLL

The EN_VSCROLL notification message is sent when the user clicks an edit control's vertical scroll bar. The control's parent window receives this notification message through a WM_COMMAND message. The parent window is notified before the screen is updated.

Parameters

wParam
Specifies the identifier of the edit control.

lParam
Specifies the handle of the edit control in the low-order word, and specifies the EN_VSCROLL notification message in the high-order word.

See Also

EN_HSCROLL, WM_COMMAND

LBN_DBLCLK

2.x

LBN_DBLCLK

The LBN_DBLCLK notification message is sent when the user double-clicks a string in a list box. The parent window of the list box receives this notification message through a WM_COMMAND message.

Parameters

wParam
Specifies the identifier of the list box.

lParam
Specifies the handle of the list box in the low-order word, and specifies the LBN_DBLCLK notification message in the high-order word.

Comments

Only a list box that has LBS_NOTIFY style will send this notification message.

See Also

LBN_SELCHANGE, WM_COMMAND

LBN_ERRSPACE

2.x

LBN_ERRSPACE

The LBN_ERRSPACE notification message is sent when a list box cannot allocate enough memory to meet a specific request. The parent window of the list box receives this notification message through a WM_COMMAND message.

Parameters

wParam
Specifies the identifier of the list box.

lParam
Specifies the handle of the list box in the low-order word, and specifies the LBN_ERRSPACE notification message in the high-order word.

See Also

WM_COMMAND

LBN_KILLFOCUS

3.0

The LBN_KILLFOCUS notification message is sent when a list box loses the input focus. The parent window of the list box receives this notification message through a WM_COMMAND message.

Parameters

wParam
Specifies the identifier of the list box.

lParam
Specifies the handle of the list box in the low-order word, and specifies the LBN_KILLFOCUS notification message in the high-order word.

See Also

LBN_SETFOCUS, WM_COMMAND

LBN_SELCANCEL

3.1

LBN_SELCANCEL

The LBN_SELCANCEL notification message is sent when the user cancels the selection in a list box. The parent window of the list box receives this notification message through a WM_COMMAND message.

Parameters

wParam
Specifies the identifier of the list box.

lParam
Specifies the handle of the list box in the low-order word, and specifies the LBN_SELCANCEL notification message in the high-order word.

Comments

This notification applies only to a list box that has the LBS_NOTIFY style.

See Also

LBN_DBLCLK, LBN_SELCHANGE, LB_SETCURSEL, WM_COMMAND

LBN_SELCHANGE

2.x

LBN_SELCHANGE

The LBN_SELCHANGE notification message is sent when the selection in a list box is about to change. The parent window of the list box receives this notification message through a WM_COMMAND message.

Parameters

wParam
Specifies the identifier of the list box.

lParam
Specifies the handle of the list box in the low-order word, and specifies the LBN_SELCHANGE notification message in the high-order word.

Comments

This notification is not sent if the selection is changed by the LB_SETCURSEL message.

This notification applies only to a list box that has the LBS_NOTIFY style.

The LBN_SELCHANGE notification is sent for a multiple-selection list box whenever the user presses an arrow key, even if the selection does not change.

See Also

LBN_DBLCLK, LBN_SELCANCEL, LB_SETCURSEL, WM_COMMAND

LBN_SETFOCUS

3.0

The LBN_SETFOCUS notification message is sent when a list box receives the input focus. The parent window of the list box receives this notification message through a WM_COMMAND message.

Parameters

wParam
Specifies the identifier of the list box.

lParam
Specifies the handle of the list box in the low-order word, and specifies the LBN_SETFOCUS notification message in the high-order word.

See Also

LBN_KILLFOCUS, WM_COMMAND

Structures

Chapter 3

This chapter defines the sizes and meanings of the structures associated with functions for the Microsoft Windows operating system, version 3.1.

Following are the Windows structures, in alphabetic order.

ABC

3.1

```
typedef struct tagABC {   /* abc */
    int   abcA;
    UINT  abcB;
    int   abcC;
} ABC;
```

The **ABC** structure contains the width of a character in a TrueType font.

Members

abcA
Specifies the "A" spacing of the character. A spacing is the distance to add to the current position before drawing the character glyph.

abcB
Specifies the "B" spacing of the character. B spacing is the width of the drawn portion of the character glyph.

abcC
Specifies the "C" spacing of the character. C spacing is the distance to add to the current position to provide white space to the right of the character glyph.

Comments

The total width of a character is the sum of the A, B, and C spaces. Either the A or the C space can be negative, to indicate underhangs or overhangs.

See Also

GetCharABCWidths

BITMAP

2.x

```
typedef struct tagBITMAP {  /* bm */
    int     bmType;
    int     bmWidth;
    int     bmHeight;
    int     bmWidthBytes;
    BYTE    bmPlanes;
    BYTE    bmBitsPixel;
    void FAR* bmBits;
} BITMAP;
```

The **BITMAP** structure defines the height, width, color format, and bit values of a logical bitmap.

Members

bmType
Specifies the bitmap type. For logical bitmaps, this member must be zero.

bmWidth
Specifies the width of the bitmap, in pixels. The width must be greater than zero.

bmHeight
Specifies the height of the bitmap, in raster lines. The height must be greater than zero.

bmWidthBytes
Specifies the number of bytes in each raster line. This value must be an even number since graphics device interface (GDI) assumes that the bit values of a bitmap form an array of integer (two-byte) values. In other words, **bmWidthBytes** * 8 must be the next multiple of 16 greater than or equal to the value obtained when the **bmWidth** member is multiplied by the **bmBitsPixel** member.

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
Points to the location of the bit values for the bitmap. The **bmBits** member must be a long pointer to an array of one-byte values.

Comments

The currently used bitmap formats are monochrome and color. The monochrome bitmap uses a one-bit, one-plane format. Each scan is a multiple of 16 bits.

Scans are organized as follows for a monochrome bitmap of height *n*:

```
Scan 0
Scan 1
.
.
.
Scan n-2
Scan n-1
```

The pixels on a monochrome device are either black or white. If the corresponding bit in the bitmap is 1, the pixel is turned on (white). If the corresponding bit in the bitmap is zero, the pixel is turned off (black).

All devices support bitmaps that have the RC_BITBLT bit set in the RASTERCAPS index of the **GetDeviceCaps** function.

Each device has its own unique color format. In order to transfer a bitmap from one device to another, use the **GetDIBits** and **SetDIBits** functions.

See Also **CreateBitmapIndirect**, **GetDIBits**, **GetObject**, **SetDIBits**

BITMAPCOREHEADER

3.0

```
typedef struct tagBITMAPCOREHEADER {    /* bmch */
    DWORD   bcSize;
    short   bcWidth;
    short   bcHeight;
    WORD    bcPlanes;
    WORD    bcBitCount;
} BITMAPCOREHEADER;
```

The **BITMAPCOREHEADER** structure contains information about the dimensions and color format of a device-independent bitmap (DIB). Windows applications should use the **BITMAPINFOHEADER** structure instead of **BITMAPCOREHEADER** whenever possible.

Members

bcSize
Specifies the number of bytes required by the **BITMAPCOREHEADER** structure.

bcWidth
Specifies the width of the bitmap, in pixels.

bcHeight
Specifies the height of the bitmap, in pixels.

bcPlanes
Specifies the number of planes for the target device. This member must be set to 1.

bcBitCount
Specifies the number of bits per pixel. This value must be 1, 4, 8, or 24.

Comments

The **BITMAPCOREINFO** structure combines the **BITMAPCOREHEADER** structure and a color table to provide a complete definition of the dimensions and colors of a DIB. See the description of the **BITMAPCOREINFO** structure for more information about specifying a DIB.

An application should use the information stored in the **bcSize** member to locate the color table in a **BITMAPCOREINFO** structure with a method such as the following:

```
pColor = ((LPSTR) pBitmapCoreInfo + (WORD) (pBitmapCoreInfo -> bcSize))
```

See Also

BITMAPCOREINFO, BITMAPINFOHEADER, BITMAPINFOHEADER

BITMAPCOREINFO

3.0

```
typedef struct tagBITMAPCOREINFO {  /* bmci */
    BITMAPCOREHEADER bmciHeader;
    RGBTRIPLE        bmciColors[1];
} BITMAPCOREINFO;
```

The **BITMAPCOREINFO** structure fully defines the dimensions and color information for a device-independent bitmap (DIB). Windows applications should use the **BITMAPINFO** structure instead of **BITMAPCOREINFO** whenever possible.

Members

bmciHeader
Specifies a **BITMAPCOREHEADER** structure that contains information about the dimensions and color format of a DIB.

bmciColors
Specifies an array of **RGBTRIPLE** structures that define the colors in the bitmap.

Comments

The **BITMAPCOREINFO** structure describes the dimensions and colors of a bitmap. It is followed immediately in memory by an array of bytes which define the pixels of the bitmap. The bits in the array are packed together, but each scan line

must be zero-padded to end on a **LONG** boundary. Segment boundaries, however, can appear anywhere in the bitmap. The origin of the bitmap is the lower-left corner.

The **bcBitCount** member of the **BITMAPCOREHEADER** structure determines the number of bits that define each pixel and the maximum number of colors in the bitmap. This member may be set to any of the following values:

Value	Meaning
1	The bitmap is monochrome, and the **bmciColors** member must contain two entries. Each bit in the bitmap array represents a pixel. If the bit is clear, the pixel is displayed with the color of the first entry in the **bmciColors** table. If the bit is set, the pixel has the color of the second entry in the table.
4	The bitmap has a maximum of 16 colors, and the **bmciColors** member contains 16 entries. Each pixel in the bitmap is represented by a four-bit index into the color table. For example, if the first byte in the bitmap is 0x1F, the byte represents two pixels. The first pixel contains the color in the second table entry, and the second pixel contains the color in the sixteenth table entry.
8	The bitmap has a maximum of 256 colors, and the **bmciColors** member contains 256 entries. In this case, each byte in the array represents a single pixel.
24	The bitmap has a maximum of 2^{24} colors. The **bmciColors** member is NULL, and each 3-byte sequence in the bitmap array represents the relative intensities of red, green, and blue, respectively, of a pixel.

The colors in the **bmciColors** table should appear in order of importance.

Alternatively, for functions that use DIBs, the **bmciColors** member can be an array of 16-bit unsigned integers that specify an index into the currently realized logical palette instead of explicit RGB values. In this case, an application using the bitmap must call DIB functions with the *wUsage* parameter set to DIB_PAL_COLORS.

Note The **bmciColors** member should not contain palette indexes if the bitmap is to be stored in a file or transferred to another application. Unless the application uses the bitmap exclusively and under its complete control, the bitmap color table should contain explicit RGB values.

See Also **BITMAPINFO, BITMAPCOREHEADER, RGBTRIPLE**

BITMAPFILEHEADER

3.0

```
typedef struct tagBITMAPFILEHEADER {    /* bmfh */
    UINT    bfType;
    DWORD   bfSize;
    UINT    bfReserved1;
    UINT    bfReserved2;
    DWORD   bfOffBits;
} BITMAPFILEHEADER;
```

The **BITMAPFILEHEADER** structure contains information about the type, size, and layout of a device-independent bitmap (DIB) file.

Members

bfType
Specifies the type of file. This member must be BM.

bfSize
Specifies the size of the file, in bytes.

bfReserved1
Reserved; must be set to zero.

bfReserved2
Reserved; must be set to zero.

bfOffBits
Specifies the byte offset from the **BITMAPFILEHEADER** structure to the actual bitmap data in the file.

Comments

A **BITMAPINFO** or **BITMAPCOREINFO** structure immediately follows the **BITMAPFILEHEADER** structure in the DIB file.

See Also

BITMAPCOREINFO, BITMAPINFO

BITMAPINFO

3.0

```
typedef struct tagBITMAPINFO {  /* bmi */
    BITMAPINFOHEADER    bmiHeader;
    RGBQUAD             bmiColors[1];
} BITMAPINFO;
```

The **BITMAPINFO** structure fully defines the dimensions and color information for a Windows 3.0 or later device-independent bitmap (DIB).

Members

bmiHeader
Specifies a **BITMAPINFOHEADER** structure that contains information about the dimensions and color format of a DIB.

bmiColors
Specifies an array of **RGBQUAD** structures that define the colors in the bitmap.

Comments

A Windows 3.0 or later DIB consists of two distinct parts: a **BITMAPINFO** structure, which describes the dimensions and colors of the bitmap, and an array of bytes defining the pixels of the bitmap. The bits in the array are packed together, but each scan line must be zero-padded to end on a **LONG** boundary. Segment boundaries, however, can appear anywhere in the bitmap. The origin of the bitmap is the lower-left corner.

The **biBitCount** member of the **BITMAPINFOHEADER** structure determines the number of bits which define each pixel and the maximum number of colors in the bitmap. This member may be set to any of the following values:

Value	Meaning
1	The bitmap is monochrome, and the **bmciColors** member must contain two entries. Each bit in the bitmap array represents a pixel. If the bit is clear, the pixel is displayed with the color of the first entry in the **bmciColors** table. If the bit is set, the pixel has the color of the second entry in the table.
4	The bitmap has a maximum of 16 colors, and the **bmciColors** member contains 16 entries. Each pixel in the bitmap is represented by a four-bit index into the color table. For example, if the first byte in the bitmap is 0x1F, the byte represents two pixels. The first pixel contains the color in the second table entry, and the second pixel contains the color in the sixteenth table entry.
8	The bitmap has a maximum of 256 colors, and the **bmciColors** member contains 256 entries. In this case, each byte in the array represents a single pixel.
24	The bitmap has a maximum of 2^{24} colors. The **bmciColors** member is NULL, and each 3-byte sequence in the bitmap array represents the relative intensities of red, green, and blue, respectively, of a pixel.

The **biClrUsed** member of the **BITMAPINFOHEADER** structure specifies the number of color indexes in the color table actually used by the bitmap. If the **biClrUsed** member is set to zero, the bitmap uses the maximum number of colors corresponding to the value of the **biBitCount** member.

The colors in the **bmiColors** table should appear in order of importance.

Alternatively, for functions that use DIBs, the **bmiColors** member can be an array of 16-bit unsigned integers that specify an index into the currently realized logical palette instead of explicit RGB values. In this case, an application using the

bitmap must call DIB functions with the *wUsage* parameter set to DIB_PAL_COLORS.

Note The **bmiColors** member should not contain palette indexes if the bitmap is to be stored in a file or transferred to another application. Unless the application uses the bitmap exclusively and under its complete control, the bitmap color table should contain explicit RGB values.

See Also **BITMAPINFOHEADER, RGBQUAD**

BITMAPINFOHEADER

3.0

```
typedef struct tagBITMAPINFOHEADER {    /* bmih */
    DWORD   biSize;
    LONG    biWidth;
    LONG    biHeight;
    WORD    biPlanes;
    WORD    biBitCount;
    DWORD   biCompression;
    DWORD   biSizeImage;
    LONG    biXPelsPerMeter;
    LONG    biYPelsPerMeter;
    DWORD   biClrUsed;
    DWORD   biClrImportant;
} BITMAPINFOHEADER;
```

The **BITMAPINFOHEADER** structure contains information about the dimensions and color format of a Windows 3.0 or later device-independent bitmap (DIB).

Members

biSize
Specifies the number of bytes required by the **BITMAPINFOHEADER** structure.

biWidth
Specifies the width of the bitmap, in pixels.

biHeight
Specifies the height of the bitmap, in pixels.

biPlanes
Specifies the number of planes for the target device. This member must be set to 1.

biBitCount
Specifies the number of bits per pixel. This value must be 1, 4, 8, or 24.

biCompression
Specifies the type of compression for a compressed bitmap. It can be one of the following values:

Value	Meaning
BI_RGB	Specifies that the bitmap is not compressed.
BI_RLE8	Specifies a run-length encoded format for bitmaps with 8 bits per pixel. The compression format is a 2-byte format consisting of a count byte followed by a byte containing a color index. For more information, see the following Comments section.
BI_RLE4	Specifies a run-length encoded format for bitmaps with 4 bits per pixel. The compression format is a 2-byte format consisting of a count byte followed by two word-length color indexes. For more information, see the following Comments section.

biSizeImage
Specifies the size, in bytes, of the image. It is valid to set this member to zero if the bitmap is in the BI_RGB format.

biXPelsPerMeter
Specifies the horizontal resolution, in pixels per meter, of the target device for the bitmap. An application can use this value to select a bitmap from a resource group that best matches the characteristics of the current device.

biYPelsPerMeter
Specifies the vertical resolution, in pixels per meter, of the target device for the bitmap.

biClrUsed
Specifies the number of color indexes in the color table actually used by the bitmap. If this value is zero, the bitmap uses the maximum number of colors corresponding to the value of the **biBitCount** member. For more information on the maximum sizes of the color table, see the description of the **BITMAPINFO** structure earlier in this chapter.

If the **biClrUsed** member is nonzero, it specifies the actual number of colors that the graphics engine or device driver will access if the **biBitCount** member is less than 24. If **biBitCount** is set to 24, **biClrUsed** specifies the size of the reference color table used to optimize performance of Windows color palettes.

If the bitmap is a packed bitmap (that is, a bitmap in which the bitmap array immediately follows the **BITMAPINFO** header and which is referenced by a single pointer), the **biClrUsed** member must be set to zero or to the actual size of the color table.

biClrImportant
Specifies the number of color indexes that are considered important for displaying the bitmap. If this value is zero, all colors are important.

Comments

The **BITMAPINFO** structure combines the **BITMAPINFOHEADER** structure and a color table to provide a complete definition of the dimensions and colors of a Windows 3.0 or later DIB. For more information about specifying a Windows 3.0 DIB, see the description of the **BITMAPINFO** structure.

An application should use the information stored in the **biSize** member to locate the color table in a **BITMAPINFO** structure as follows:

```
pColor = ((LPSTR) pBitmapInfo + (WORD) (pBitmapInfo->bmiHeader.biSize))
```

Windows supports formats for compressing bitmaps that define their colors with 8 bits per pixel and with 4 bits per pixel. Compression reduces the disk and memory storage required for the bitmap. The following paragraphs describe these formats.

BI_RLE8 When the **biCompression** member is set to BI_RLE8, the bitmap is compressed using a run-length encoding format for an 8-bit bitmap. This format may be compressed in either of two modes: encoded and absolute. Both modes can occur anywhere throughout a single bitmap.

Encoded mode consists of two bytes: the first byte specifies the number of consecutive pixels to be drawn using the color index contained in the second byte. In addition, the first byte of the pair can be set to zero to indicate an escape that denotes an end of line, end of bitmap, or a delta. The interpretation of the escape depends on the value of the second byte of the pair. The following list shows the meaning of the second byte:

Value	Meaning
0	End of line.
1	End of bitmap.
2	Delta. The two bytes following the escape contain unsigned values indicating the horizontal and vertical offset of the next pixel from the current position.

Absolute mode is signaled by the first byte set to zero and the second byte set to a value between 0x03 and 0xFF. In absolute mode, the second byte represents the number of bytes that follow, each of which contains the color index of a single pixel. When the second byte is set to 2 or less, the escape has the same meaning as in encoded mode. In absolute mode, each run must be aligned on a word boundary.

The following example shows the hexadecimal values of an 8-bit compressed bitmap:

```
03 04 05 06 00 03 45 56 67 00 02 78 00 02 05 01
02 78 00 00 09 1E 00 01
```

This bitmap would expand as follows (two-digit values represent a color index for a single pixel):

```
04 04 04
06 06 06 06 06
45 56 67
78 78
move current position 5 right and 1 down
78 78
end of line
1E 1E 1E 1E 1E 1E 1E 1E
end of RLE bitmap
```

BI_RLE4 When the **biCompression** member is set to BI_RLE4, the bitmap is compressed using a run-length encoding (RLE) format for a 4-bit bitmap, which also uses encoded and absolute modes. In encoded mode, the first byte of the pair contains the number of pixels to be drawn using the color indexes in the second byte. The second byte contains two color indexes, one in its high-order nibble (that is, its low-order four bits) and one in its low-order nibble. The first of the pixels is drawn using the color specified by the high-order nibble, the second is drawn using the color in the low-order nibble, the third is drawn with the color in the high-order nibble, and so on, until all the pixels specified by the first byte have been drawn.

In absolute mode, the first byte contains zero, the second byte contains the number of color indexes that follow, and subsequent bytes contain color indexes in their high- and low-order nibbles, one color index for each pixel. In absolute mode, each run must be aligned on a word boundary. The end-of-line, end-of-bitmap, and delta escapes also apply to BI_RLE4.

The following example shows the hexadecimal values of a 4-bit compressed bitmap:

```
03 04 05 06 00 06 45 56 67 00 04 78 00 02 05 01
04 78 00 00 09 1E 00 01
```

This bitmap would expand as follows (single-digit values represent a color index for a single pixel):

```
0 4 0
0 6 0 6 0
4 5 5 6 6 7
7 8 7 8
move current position 5 right and 1 down
7 8 7 8
end of line
1 E 1 E 1 E 1 E 1
end of RLE bitmap
```

See Also **BITMAPINFO**

CBT_CREATEWND

3.1

```
typedef struct tagCBT_CREATEWND {   /* cbtcw */
    CREATESTRUCT FAR* lpcs;
    HWND              hwndInsertAfter;
} CBT_CREATEWND;
```

The **CBT_CREATEWND** structure contains information passed to a WH_CBT hook function before a window is created.

Members

lpcs
Points to a **CREATESTRUCT** structure that contains initialization parameters for the window about to be created.

hwndInsertAfter
Identifies a window in the window manager's list that will precede the window being created. If this parameter is NULL, the window being created is the top-most window. If this parameter is 1, the window being created is the bottom-most window.

See Also

CBTProc, **SetWindowsHook**

CBTACTIVATESTRUCT

3.1

```
typedef struct tagCBTACTIVATESTRUCT { /* cas */
    BOOL    fMouse;
    HWND    hWndActive;
} CBTACTIVATESTRUCT;
```

The **CBTACTIVATESTRUCT** structure contains information passed to a WH_CBT hook function before a window is activated.

Members

fMouse
Specifies whether the window is being activated as a result of a mouse click. This value is nonzero if a mouse click is causing the activation. Otherwise, this value is zero.

hWndActive
Identifies the currently active window.

See Also

SetWindowsHook

CHOOSECOLOR

3.1

```
#include <commdlg.h>

typedef struct tagCHOOSECOLOR {    /* cc */
    DWORD    lStructSize;
    HWND     hwndOwner;
    HWND     hInstance;
    COLORREF rgbResult;
    COLORREF FAR* lpCustColors;
    DWORD    Flags;
    LPARAM   lCustData;
    UINT     (CALLBACK* lpfnHook)(HWND, UINT, WPARAM, LPARAM);
    LPCSTR   lpTemplateName;
} CHOOSECOLOR;
```

The **CHOOSECOLOR** structure contains information that the system uses to initialize the system-defined Color dialog box. After the user chooses the OK button to close the dialog box, the system returns information about the user's selection in this structure.

Members

lStructSize
Specifies the length of the structure, in bytes. This member is filled on input.

hwndOwner
Identifies the window that owns the dialog box. This member can be any valid window handle, or it should be NULL if the dialog box is to have no owner.

If the CC_SHOWHELP flag is set, **hwndOwner** must identify the window that owns the dialog box. The window procedure for this owner window receives a notification message when the user chooses the Help button. (The identifier for the notification message is the value returned by the **RegisterWindowMessage** function when HELPMSGSTRING is passed as its argument.)

This member is filled on input.

hInstance
Identifies a data block that contains the dialog box template specified by the **lpTemplateName** member. This member is used only if the **Flags** member specifies the CC_ENABLETEMPLATE or CC_ENABLETEMPLATEHANDLE flag; otherwise, this member is ignored. This member is filled on input.

rgbResult
Specifies the color that is initially selected when the dialog box is displayed, and specifies the user's color selection after the user has chosen the OK button to close dialog box. If the CC_RGBINIT flag is set in the **Flags** member before the dialog box is displayed and the value of this member is not among the colors available, the system selects the nearest solid color available. If this

member is NULL, the first selected color is black. This member is filled on input and output.

lpCustColors

Points to an array of 16 doubleword values, each of which specifies the intensities of the red, green, and blue (RGB) components of a custom color box in the dialog box. If the user modifies a color, the system updates the array with the new RGB values. This member is filled on input and output.

Flags

Specifies the dialog box initialization flags. This member may be a combination of the following values:

Value	Meaning
CC_ENABLEHOOK	Enables the hook function specified in the **lpfnHook** member.
CC_ENABLETEMPLATE	Causes the system to use the dialog box template identified by the **hInstance** member and pointed to by the **lpTemplateName** member.
CC_ENABLETEMPLATEHANDLE	Indicates that the **hInstance** member identifies a data block that contains a preloaded dialog box template. If this flag is specified, the system ignores the **lpTemplateName** member.
CC_FULLOPEN	Causes the entire dialog box to appear when the dialog box is displayed, including the portion that allows the user to create custom colors. Without this flag, the user must select the Define Custom Color button to see that portion of the dialog box.
CC_PREVENTFULLOPEN	Disables the Define Custom Colors button, preventing the user from creating custom colors.
CC_RGBINIT	Causes the dialog box to use the color specified in the **rgbResult** member as the initial color selection.
CC_SHOWHELP	Causes the dialog box to show the Help button. If this flag is specified, the **hwndOwner** member must not be NULL.

These flags are used when the structure is initialized.

lCustData
Specifies application-defined data that the system passes to the hook function pointed to by the **lpfnHook** member. The system passes a pointer to the **CHOOSECOLOR** structure in the *lParam* parameter of the WM_INITDIALOG message; this pointer can be used to retrieve the **lCustData** member.

lpfnHook
Points to a hook function that processes messages intended for the dialog box. To enable the hook function, an application must specify the CC_ENABLEHOOK value in the **Flags** member; otherwise, the system ignores this structure member. The hook function must return zero to pass a message that it didn't process back to the dialog box procedure in COMMDLG.DLL. The hook function must return a nonzero value to prevent the dialog box procedure in COMMDLG.DLL from processing a message it has already processed. This member is filled on input.

lpTemplateName
Points to a null-terminated string that specifies the name of the resource file for the dialog box template that is to be substituted for the dialog box template in COMMDLG.DLL. An application can use the **MAKEINTRESOURCE** macro for numbered dialog box resources. This member is used only if the **Flags** member specifies the CC_ENABLETEMPLATE flag; otherwise, this member is ignored. This member is filled on input.

Comments

Some members of this structure are filled only when the dialog box is created, and some have an initialization value that changes when the user closes the dialog box. Whenever a description in the Members section does not specify how the value of a member is assigned, the value is assigned only when the dialog box is created.

See Also

ChooseColor

CHOOSEFONT

3.1

```
#include <commdlg.h>

typedef struct tagCHOOSEFONT {  /* cf */
    DWORD           lStructSize;
    HWND            hwndOwner;
    HDC             hDC;
    LOGFONT FAR*    lpLogFont;
    int             iPointSize;
    DWORD           Flags;
    COLORREF        rgbColors;
    LPARAM          lCustData;
    UINT (CALLBACK* lpfnHook)(HWND, UINT, WPARAM, LPARAM);
    LPCSTR          lpTemplateName;
    HINSTANCE       hInstance;
    LPSTR           lpszStyle;
    UINT            nFontType;
    int             nSizeMin;
    int             nSizeMax;

} CHOOSEFONT;
```

The **CHOOSEFONT** structure contains information that the system uses to initialize the system-defined Font dialog box. After the user chooses the OK button to close the dialog box, the system returns information about the user's selection in this structure.

Members

lStructSize
Specifies the length of the structure, in bytes. This member is filled on input.

hwndOwner
Identifies the window that owns the dialog box. This member can be any valid window handle, or it should be NULL if the dialog box is to have no owner.

If the CF_SHOWHELP flag is set, **hwndOwner** must identify the window that owns the dialog box. The window procedure for this owner window receives a notification message when the user chooses the Help button. (The identifier for the notification message is the value returned by the **RegisterWindowMessage** function when HELPMSGSTRING is passed as its argument.)

This member is filled on input.

hDC
Identifies either the device context or the information context of the printer for which fonts are to be listed in the dialog box. This member is used only if the **Flags** member specifies the CF_PRINTERFONTS flag; otherwise, this member is ignored.

This member is filled on input.

lpLogFont
Points to a **LOGFONT** structure. If an application initializes the members of this structure before calling **ChooseFont** and sets the CF_INITTOLOGFONTSTRUCT flag, the **ChooseFont** function initializes the dialog box with the font that is the closest possible match. After the user chooses the OK button to close the dialog box, the **ChooseFont** function sets the members of the **LOGFONT** structure based on the user's final selection.

This member is filled on input and output.

iPointSize
Specifies the size of the selected font, in tenths of a point. The **ChooseFont** function sets this value after the user chooses the OK button to close the dialog box.

Flags
Specifies the dialog box initialization flags. This member can be a combination of the following values:

Value	Meaning
CF_APPLY	Specifies that the **ChooseFont** function should enable the Apply button.
CF_ANSIONLY	Specifies that the **ChooseFont** function should limit font selection to those fonts that use the Windows character set. (If this flag is set, the user cannot select a font that contains only symbols.)
CF_BOTH	Causes the dialog box to list the available printer and screen fonts. The **hDC** member identifies either the device context or the information context associated with the printer.
CF_TTONLY	Specifies that the **ChooseFont** function should enumerate and allow the selection of only TrueType fonts.
CF_EFFECTS	Specifies that the **ChooseFont** function should enable strikeout, underline, and color effects. If this flag is set, the **lfStrikeOut** and **lfUnderline** members of the **LOGFONT** structure and the **rgbColors** member of the **CHOOSEFONT** structure can be set before calling **ChooseFont**. And, if this flag is not set, the **ChooseFont** function can set these members after the user chooses the OK button to close the dialog box.

Value	Meaning
CF_ENABLEHOOK	Enables the hook function specified in the **lpfnHook** member of this structure.
CF_ENABLETEMPLATE	Indicates that the **hInstance** member identifies a data block that contains the dialog box template pointed to by **lpTemplateName**.
CF_ENABLETEMPLATEHANDLE	Indicates that the **hInstance** member identifies a data block that contains a pre-loaded dialog box template. If this flag is specified, the system ignores the **lpTemplateName** member.
CF_FIXEDPITCHONLY	Specifies that the **ChooseFont** function should select only monospace fonts.
CF_FORCEFONTEXIST	Specifies that the **ChooseFont** function should indicate an error condition if the user attempts to select a font or font style that does not exist.
CF_INITTOLOGFONTSTRUCT	Specifies that the **ChooseFont** function should use the **LOGFONT** structure pointed to by **lpLogFont** to initialize the dialog box controls.
CF_LIMITSIZE	Specifies that the **ChooseFont** function should select only font sizes within the range specified by the **nSizeMin** and **nSizeMax** members.
CF_NOFACESEL	Specifies that there is no selection in the Font (face name) combo box. Applications use this flag to support multiple font selections. This flag is set on input and output.
CF_NOOEMFONTS	Specifies that the **ChooseFont** function should not allow vector-font selections. This flag has the same value as CF_NOVECTORFONTS.
CF_NOSIMULATIONS	Specifies that the **ChooseFont** function should not allow graphics-device-interface (GDI) font simulations.
CF_NOSIZESEL	Specifies that there is no selection in the Size combo box. Applications use this flag to support multiple size selections. This flag is set on input and output.
CF_NOSTYLESEL	Specifies that there is no selection in the Font Style combo box. Applications use this flag to support multiple style selections. This flag is set on input and output.

Value	Meaning
CF_NOVECTORFONTS	Specifies that the **ChooseFont** function should not allow vector-font selections. This flag has the same value as CF_NOOEMFONTS.
CF_PRINTERFONTS	Causes the dialog box to list only the fonts supported by the printer associated with the device context or information context that is identified by the **hDC** member.
CF_SCALABLEONLY	Specifies that the **ChooseFont** function should allow the selection of only scalable fonts. (Scalable fonts include vector fonts, some printer fonts, TrueType fonts, and fonts that are scaled by other algorithms or technologies.)
CF_SCREENFONTS	Causes the dialog box to list only the screen fonts supported by the system.
CF_SHOWHELP	Causes the dialog box to show the Help button. If this option is specified, the **hwndOwner** must not be NULL.
CF_USESTYLE	Specifies that the **lpszStyle** member points to a buffer that contains a style-description string that the **ChooseFont** function should use to initialize the Font Style box. When the user chooses the OK button to close the dialog box, the **ChooseFont** function copies the style description for the user's selection to this buffer.
CF_WYSIWYG	Specifies that the **ChooseFont** function should allow the selection of only fonts that are available on both the printer and the screen. If this flag is set, the CF_BOTH and CF_SCALABLEONLY flags should also be set.

These flags may be set when the structure is initialized, except where specified.

rgbColors

If the CF_EFFECTS flag is set, this member contains the red, green, and blue (RGB) values the **ChooseFont** function should use to set the text color. After the user chooses the OK button to close the dialog box, this member contains the RGB values of the color the user selected.

This member is filled on input and output.

lCustData
Specifies application-defined data that the application passes to the hook function. The system passes a pointer to the CHOOSEFONT data structure in the *lParam* parameter of the WM_INITDIALOG message; the **lCustData** member can be retrieved using this pointer.

lpfnHook
Points to a hook function that processes messages intended for the dialog box. To enable the hook function, an application must specify the CF_ENABLEHOOK value in the **Flags** member; otherwise, the system ignores this structure member. The hook function must return zero to pass a message that it didn't process back to the dialog box procedure in COMMDLG.DLL. The hook function must return a nonzero value to prevent the dialog box procedure in COMMDLG.DLL from processing a message it has already processed.

This member is filled on input.

lpTemplateName
Points to a null-terminated string that specifies the name of the resource file for the dialog box template to be substituted for the dialog box template in COMMDLG.DLL. An application can use the **MAKEINTRESOURCE** macro for numbered dialog box resources. This member is used only if the **Flags** member specifies the CF_ENABLETEMPLATE flag; otherwise, this member is ignored.

This member is filled on input.

hInstance
Identifies a data block that contains the dialog box template specified by the **lpTemplateName** member. This member is used only if the **Flags** member specifies the CF_ENABLETEMPLATE or the CF_ENABLETEMPLATEHANDLE flag; otherwise, this member is ignored.

This member is filled on input.

lpszStyle
Points to a buffer that contains a style-description string for the font. If the CF_USESTYLE flag is set, the **ChooseFont** function uses the data in this buffer to initialize the Font Style box. When the user chooses the OK button to close the dialog box, the **ChooseFont** function copies the string in the Font Style box into this buffer.

The buffer pointed to by **lpszStyle** must be at least LF_FACESIZE bytes long.

This member is filled on input and output.

nFontType

Specifies the type of the selected font. This member can be one or more of the values in the following list:

Value	Meaning
BOLD_FONTTYPE	Specifies that the font is bold. This value applies only to TrueType fonts. This value corresponds to the value of the **ntmFlags** member of the **NEWTEXTMETRIC** structure.
ITALIC_FONTTYPE	Specifies that the font is italic. This value applies only to TrueType fonts. This value corresponds to the value of the **ntmFlags** member of the **NEWTEXTMETRIC** structure.
PRINTER_FONTTYPE	Specifies that the font is a printer font.
REGULAR_FONTTYPE	Specifies that the font is neither bold nor italic. This value applies only to TrueType fonts. This value corresponds to the value of the **ntmFlags** member of the **NEWTEXTMETRIC** structure.
SCREEN_FONTTYPE	Specifies that the font is a screen font.
SIMULATED_FONTTYPE	Specifies that the font is simulated by GDI. This is not set if the CF_NOSIMULATIONS flag is set.

nSizeMin

Specifies the minimum point size that a user can select. The **ChooseFont** function will recognize this member only if the CF_LIMITSIZE flag is set.

This member is filled on input.

nSizeMax

Specifies the maximum point size that a user can select. The **ChooseFont** function will recognize this member only if the CF_LIMITSIZE flag is set.

This member is filled on input.

See Also **ChooseFont**

CLASSENTRY

3.1

```
#include <toolhelp.h>

typedef struct tagCLASSENTRY {  /* ce */
    DWORD   dwSize;
    HMODULE hInst;
    char    szClassName[MAX_CLASSNAME + 1];
    WORD    wNext;
} CLASSENTRY;
```

The **CLASSENTRY** structure contains the name of a Windows class and a near pointer to the next class in the list. For more information about Windows classes, see the **GetClassInfo** function in the *Microsoft Windows Programmer's Reference, Volume 2*.

Members

dwSize
Specifies the size of the **CLASSENTRY** structure, in bytes.

hInst
Identifies the instance handle of the task that owns the class. An application needs this handle to call **GetClassInfo**. The **hInst** member is really a handle to a module, since Windows classes are owned by modules. Therefore, this **hInst** will not match the **hInst** passed as a parameter to the **WinMain** function of the owning task.

szClassName
Specifies the null-terminated string that contains the class name. An application needs this name to call **GetClassInfo**.

wNext
Specifies the next class in the list. This member is reserved for internal use by Windows.

See Also

ClassFirst, **ClassNext**

CLIENTCREATESTRUCT

3.0

```
typedef struct tagCLIENTCREATESTRUCT {  /* ccs */
    HANDLE  hWindowMenu;
    UINT    idFirstChild;
} CLIENTCREATESTRUCT;
```

The **CLIENTCREATESTRUCT** structure contains information about the menu and first multiple document interface (MDI) child window of an MDI client window. An application passes a long pointer to this structure as the *lpParam* parameter of the **CreateWindow** function when creating an MDI client window.

Members

hWindowMenu

Identifies the menu handle of the application's Window menu. An application can retrieve this handle from the menu of the MDI frame window by using the **GetSubMenu** function.

idFirstChild

Specifies the child window identifier of the first MDI child window created. Windows increments the identifier for each additional MDI child window that the application creates, and reassigns identifiers when the application destroys a window to keep the range of identifiers continuous. These identifiers are used in WM_COMMAND messages to the application's MDI frame window when a child window is selected from the Window menu; they should not conflict with any other command identifiers.

See Also

CreateWindow, **GetSubMenu**

COMPAREITEMSTRUCT

3.0

```
typedef struct tagCOMPAREITEMSTRUCT {    /* cis */
    UINT  CtlType;
    UINT  CtlID;
    HWND  hwndItem;
    UINT  itemID1;
    DWORD itemData1;
    UINT  itemID2;
    DWORD itemData2;
} COMPAREITEMSTRUCT;
```

The **COMPAREITEMSTRUCT** structure supplies the identifiers and application-supplied data for two items in a sorted owner-drawn combo box or list box.

Whenever an application adds a new item to an owner-drawn combo or list box created with the CBS_SORT or LBS_SORT style, Windows sends the owner a WM_COMPAREITEM message. The *lParam* parameter of the message contains a long pointer to a **COMPAREITEMSTRUCT** structure. When the owner receives the message, it compares the two items and returns a value indicating which item sorts before the other. For more information, see the description of the WM_COMPAREITEM message in Chapter 2, "Messages."

Members

CtlType
Specifies ODT_LISTBOX (which identifies an owner-drawn list box) or ODT_COMBOBOX (which identifies an owner-drawn combo box).

CtlID
Specifies the identifier of the list box or combo box.

hwndItem
Identifies the control.

itemID1
Specifies the index of the first item in the list box or combo box being compared.

itemData1
Specifies application-supplied data for the first item being compared. (This value was passed as the *lParam* parameter of the message that added the item to the combo box or list box.)

itemID2
Specifies the index of the second item in the list box or combo box being compared.

itemData2
Specifies application-supplied data for the second item being compared. This value was passed as the *lParam* parameter of the message that added the item to the combo box or list box.

COMSTAT

3.1

```
typedef struct tagCOMSTAT {    /* cmst                              */
    BYTE status;               /* status of transmission            */
    UINT cbInQue;              /* count of characters in Rx Queue */
    UINT cbOutQue;             /* count of characters in Tx Queue */
} COMSTAT;
```

The **COMSTAT** structure contains information about a communications device.

Members

status

Specifies the status of the transmission. This member can be one or more of the following flags:

Flag	Meaning
CSTF_CTSHOLD	Specifies whether transmission is waiting for the CTS (clear-to-send) signal to be sent.
CSTF_DSRHOLD	Specifies whether transmission is waiting for the DSR (data-set-ready) signal to be sent.
CSTF_RLSDHOLD	Specifies whether transmission is waiting for the RLSD (receive-line-signal-detect) signal to be sent.
CSTF_XOFFHOLD	Specifies whether transmission is waiting as a result of the XOFF character being received.
CSTF_XOFFSENT	Specifies whether transmission is waiting as a result of the XOFF character being transmitted. Transmission halts when the XOFF character is transmitted and used by systems that take the next character as XON, regardless of the actual character.
CSTF_EOF	Specifies whether the end-of-file (EOF) character has been received.
CSTF_TXIM	Specifies whether a character is waiting to be transmitted.

cbInQue

Specifies the number of characters in the receive queue.

cbOutQue

Specifies the number of characters in the transmit queue.

See Also

GetCommError

CONVCONTEXT

3.1

```
#include <ddeml.h>

typedef struct tagCONVCONTEXT { /* cc                              */
    UINT        cb;
    UINT        wFlags;
    UINT        wCountryID;
    int         iCodePage;
    DWORD       dwLangID;
    DWORD       dwSecurity;
} CONVCONTEXT;
```

The **CONVCONTEXT** structure contains information that makes it possible for applications to share data in several different languages.

Members

cb
Specifies the size, in bytes, of the **CONVCONTEXT** structure.

wFlags
Specifies conversation-context flags. Currently, no flags are defined for this member.

wCountryID
Specifies the country-code identifier for topic-name and item-name strings.

iCodePage
Specifies the code page for topic-name and item-name strings. Unilingual clients should set this member to CP_WINANSI. An application that uses the OEM character set should set this member to the value returned by the **GetKBCodePage** function. For more information about the OEM character set, see the *Microsoft Windows Guide to Programming*.

dwLangID
Specifies the language identifier for topic-name and item-name strings.

dwSecurity
Specifies a private (application-defined) security code.

See Also

GetKBCodePage

CONVINFO

3.1

```
#include <ddeml.h>

typedef struct tagCONVINFO { /* ci */
    DWORD    cb;
    DWORD    hUser;
    HCONV    hConvPartner;
    HSZ      hszSvcPartner;
    HSZ      hszServiceReq;
    HSZ      hszTopic;
    HSZ      hszItem;
    UINT     wFmt;
    UINT     wType;
    UINT     wStatus;
    UINT     wConvst;
    UINT     wLastError;
    HCONVLIST hConvList;
    CONVCONTEXT ConvCtxt;
} CONVINFO;
```

The **CONVINFO** structure contains information about a dynamic data exchange (DDE) conversation.

Members

cb
Specifies the length of the structure, in bytes.

hUser
Identifies application-defined data.

hConvPartner
Identifies the partner application in the DDE conversation. If the partner has not registered itself (by using the **DdeInitialize** function) to make DDE Management Library (DDEML) function calls, this member is set to 0. An application should not pass this member to any DDEML function except **DdeQueryConvInfo**.

hszSvcPartner
Identifies the service name of the partner application.

hszServiceReq
Identifies the service name of the server application that was requested for connection.

hszTopic
Identifies the name of the requested topic.

hszItem
Identifies the name of the requested item. This member is transaction-specific.

wFmt
Specifies the format of the data being exchanged. This member is transaction-specific.

wType
Specifies the type of the current transaction. This member is transaction-specific and can be one of the following values:

Value	Meaning
XTYP_ADVDATA	Informs a client that advise data from a server has arrived.
XTYP_ADVREQ	Requests that a server send updated data to the client during an advise loop. This transaction results when the server calls the **DdePostAdvise** function.
XTYP_ADVSTART	Requests that a server begin an advise loop with a client.
XTYP_ADVSTOP	Notifies a server that an advise loop is ending.
XTYP_CONNECT	Requests that a server establish a conversation with a client.
XTYP_CONNECT_CONFIRM	Notifies a server that a conversation with a client has been established.
XTYP_DISCONNECT	Notifies a server that a conversation has terminated.
XTYP_ERROR	Notifies a DDEML application that a critical error has occurred. The DDEML may have insufficient resources to continue.
XTYP_EXECUTE	Requests that a server execute a command sent by a client.
XTYP_MONITOR	Notifies an application registered as APPCMD_MONITOR of DDE data being transmitted.
XTYP_POKE	Requests that a server accept unsolicited data from a client.
XTYP_REGISTER	Notifies other DDEML applications that a server has registered a service name.
XTYP_REQUEST	Requests that a server send data to a client.
XTYP_UNREGISTER	Notifies other DDEML applications that a server has unregistered a service name.
XTYP_WILDCONNECT	Requests that a server establish multiple conversations with the same client.
XTYP_XACT_COMPLETE	Notifies a client that an asynchronous data transaction has completed.

wStatus
Specifies the status of the current conversation. This member can be a combination of the following values:

ST_ADVISE	ST_INLIST
ST_BLOCKED	ST_ISLOCAL
ST_BLOCKNEXT	ST_ISSELF
ST_CLIENT	ST_TERMINATED
ST_CONNECTED	

wConvst
Specifies the conversation state. This member can be one of the following values:

XST_ADVACKRCVD	XST_INIT1
XST_ADVDATAACKRCVD	XST_INIT2
XST_ADVDATASENT	XST_NULL
XST_ADVSENT	XST_POKEACKRCVD
XST_CONNECTED	XST_POKESENT
XST_DATARCVD	XST_REQSENT
XST_EXECACKRCVD	XST_UNADVACKRCVD
XST_EXECSENT	XST_UNADVSENT
XST_INCOMPLETE	

wLastError
Specifies the error value associated with the last transaction.

hConvList
If the handle of the current conversation is in a conversation list, identifies the conversation list. Otherwise, this member is NULL.

ConvCtxt
Specifies the conversation context.

See Also CONVCONTEXT

CPLINFO

3.1

```
#include <cpl.h>

typedef struct tagCPLINFO { /* cpli */
    int   idIcon;
    int   idName;
    int   idInfo;
    LONG  lData;
} CPLINFO;
```

The **CPLINFO** structure contains resource information and a user-defined value for an extensible Control Panel application.

Members

idIcon
Specifies an icon resource identifier for the application icon. This icon is displayed in the Control Panel window.

idName
Specifies a string resource identifier for the application name. The name is the short string displayed below the application icon in the Control Panel window. The name is also displayed on the Settings menu of Control Panel.

idInfo
Specifies a string resource identifier for the application description. The description is the descriptive string displayed at the bottom of the Control Panel window when the application icon is selected.

lData
Specifies user-defined data for the application.

CREATESTRUCT

2.x

```
typedef struct tagCREATESTRUCT {    /* cs */
    void FAR* lpCreateParams;
    HINSTANCE hInstance;
    HMENU     hMenu;
    HWND      hwndParent;
    int       cy;
    int       cx;
    int       y;
    int       x;
    LONG      style;
    LPCSTR    lpszName;
    LPCSTR    lpszClass;
    DWORD     dwExStyle;
} CREATESTRUCT;
```

The **CREATESTRUCT** structure defines the initialization parameters passed to the window procedure of an application.

Members

lpCreateParams
Points to data to be used for creating the window.

hInstance
Identifies the module-instance handle of the module that owns the new window.

hMenu
Identifies the menu to be used by the new window.

hwndParent
Identifies the window that owns the new window. This member is NULL if the new window is a top-level window.

cy
Specifies the height of the new window.

cx
Specifies the width of the new window.

y
Specifies the y-coordinate of the upper-left corner of the new window. Coordinates are relative to the parent window if the new window is a child window. Otherwise, the coordinates are relative to the screen origin.

x
Specifies the x-coordinate of the upper-left corner of the new window. Coordinates are relative to the parent window if the new window is a child window. Otherwise, the coordinates are relative to the screen origin.

style
Specifies the style for the new window.

lpszName
Points to a null-terminated string that specifies the name of the new window.

lpszClass
Points to a null-terminated string that specifies the class name of the new window.

dwExStyle
Specifies extended style for the new window.

See Also **CreateWindow**

CTLINFO

3.1

```
#include <custcntl.h>

typedef struct tagCTLINFO {
    UINT    wVersion;                    /* control version */
    UINT    wCtlTypes;                   /* control types   */
    char    szClass[CTLCLASS];         /* control class name */
    char    szTitle[CTLTITLE];         /* control title      */
    char    szReserved[10];          /* reserved for future use */
    CTLTYPE Type[CTLTYPES];          /* control type list       */
} CTLINFO;
```

The **CTLINFO** structure defines the class name and version number for a custom control. The **CTLINFO** structure also contains an array of **CTLTYPE** structures, each of which lists commonly used combinations of control styles (called variants), with a short description and information about the suggested size.

Members

wVersion
Specifies the control version number. Although you can start your numbering scheme from one digit, most implementations use the lower two digits to represent minor releases.

wCtlTypes
Specifies the number of control types supported by this class. This value should always be greater than zero and less than or equal to the **CTLTYPES** value.

szClass
Specifies a null-terminated string that contains the control class name supported by the dynamic-link library (DLL). This string should be no longer than the **CTLCLASS** value.

szTitle
Specifies a null-terminated string that contains various copyright or author information relating to the control library. This string should be no longer than the **CTLTITLE** value.

Type
Specifies an array of **CTLTYPE** structures containing information that relates to each of the control types supported by the class. There should be no more elements in the array than specified by the **CTLTYPES** value.

Comments

An application calls the *Class***Info** function to retrieve basic information about the control library. Based on the information returned, the application can create instances of a control by using one of the supported styles. For example, Dialog Editor calls this function to query a library about the different control styles it can display.

The return value of the *Class***Info** function identifies a **CTLINFO** structure if the function is successful. This information becomes the property of the caller, which must explicitly release it by using the **GlobalFree** function when the structure is no longer needed.

See Also

CTLSTYLE, CTLTYPE

CTLSTYLE

3.1

```
#include <custcntl.h>

typedef struct tagCTLSTYLE {
    UINT    wX;                 /* x-origin of control   */
    UINT    wY;                 /* y-origin of control   */
    UINT    wCx;                /* width of control      */
    UINT    wCy;                /* height of control     */
    UINT    wId;                /* control child id      */
    DWORD   dwStyle;            /* control style         */
    char    szClass[CTLCLASS];  /* name of control class */
    char    szTitle[CTLTITLE];  /* control text          */
} CTLSTYLE;
```

The **CTLSTYLE** structure specifies the attributes of the selected control, including the current style flags, location, dimensions, and associated text.

Members

wX
Specifies the x-origin, in screen coordinates, of the control relative to the client area of the parent window.

wY
Specifies the y-origin, in screen coordinates, of the control relative to the client area of the parent window.

wCx
Specifies the current control width, in screen coordinates.

wCy
Specifies the current control height, in screen coordinates.

wId
Specifies the current control identifier. In most cases, you should not allow the user to change this value because Dialog Editor automatically coordinates it with a header file.

dwStyle
Specifies the current control style. The high-order word contains the control-specific flags, and the low-order word contains the Windows-specific flags. You may let the user change these flags to any values supported by your control library.

szClass
Specifies a null-terminated string representing the name of the current control class. You should not allow the user to edit this member, because it is provided for informational purposes only. This string should be no longer than the **CTLCLASS** value.

szTitle
Specifies with a null-terminated string the text associated with the control. This text is usually displayed inside the control or may be used to store other associated information required by the control. This string should be no longer than the **CTLTITLE** value.

Comments

An application calls the *Class***Style** function to display a dialog box to edit the style of the selected control. When this function is called, it should display a modal dialog box in which the user can edit the **CTLSTYLE** members. The user interface of this dialog box should be consistent with that of the predefined controls that Dialog Editor supports.

See Also

CTLINFO, CTLTYPE

CTLTYPE

3.1

```
#include <custcntl.h>

typedef struct tagCTLTYPE {
    UINT    wType;              /* type style       */
    UINT    wWidth;             /* suggested width  */
    UINT    wHeight;            /* suggested height */
    DWORD   dwStyle;            /* default style    */
    char    szDescr[CTLDESCR];  /* menu name        */
} CTLTYPE;
```

The **CTLTYPE** structure contains information about a control in a particular class. The **CTLINFO** structure includes an array of **CTLTYPE** structures.

Members

wType
Reserved; must be zero.

wWidth
Specifies the suggested width of the control when created with Dialog Editor. The width is specified in resource-compiler coordinates.

wHeight
Specifies the suggested height of the control when created using Dialog Editor. The height is specified in resource-compiler coordinates.

dwStyle
Specifies the initial style bits used to obtain this control type. This value includes the control-defined flags in the high-order word and the Windows-defined flags in the low-order word.

szDescr
Defines the name to be used by other development tools when referring to this particular variant of the base control class. Dialog Editor does not refer to this information. This string should not be longer than the **CTLDESCR** value.

See Also

CTLINFO, **CTLSTYLE**

DCB

2.x

```
typedef struct tagDCB        /* dcb                              */
{
    BYTE Id;                 /* internal device identifier       */
    UINT BaudRate;           /* baud rate                        */
    BYTE ByteSize;           /* number of bits/byte, 4-8         */
    BYTE Parity;             /* 0-4=none,odd,even,mark,space     */
    BYTE StopBits;           /* 0,1,2 = 1, 1.5, 2                */
    UINT RlsTimeout;         /* timeout for RLSD to be set       */
    UINT CtsTimeout;         /* timeout for CTS to be set        */
    UINT DsrTimeout;         /* timeout for DSR to be set        */

    UINT fBinary        :1;  /* binary mode (skip EOF check)     */
    UINT fRtsDisable    :1;  /* don't assert RTS at init time    */
    UINT fParity        :1;  /* enable parity checking           */
    UINT fOutxCtsFlow   :1;  /* CTS handshaking on output        */
    UINT fOutxDsrFlow   :1;  /* DSR handshaking on output        */
    UINT fDummy         :2;  /* reserved                         */
    UINT fDtrDisable    :1;  /* don't assert DTR at init time    */

    UINT fOutX          :1;  /* enable output XON/XOFF           */
    UINT fInX           :1;  /* enable input XON/XOFF            */
    UINT fPeChar        :1;  /* enable parity err replacement    */
    UINT fNull          :1;  /* enable null stripping            */
    UINT fChEvt         :1;  /* enable Rx character event        */
    UINT fDtrflow       :1;  /* DTR handshake on input           */
    UINT fRtsflow       :1;  /* RTS handshake on input           */
    UINT fDummy2        :1;

    char XonChar;            /* Tx and Rx XON character          */
    char XoffChar;           /* Tx and Rx XOFF character         */
    UINT XonLim;             /* transmit XON threshold           */
    UINT XoffLim;            /* transmit XOFF threshold          */
    char PeChar;             /* parity error replacement char    */
    char EofChar;            /* end of Input character           */
    char EvtChar;            /* received event character         */
    UINT TxDelay;            /* amount of time between chars     */
} DCB;
```

The **DCB** structure defines the control setting for a serial communications device.

Members

Id

Specifies the communication device. This value is set by the device driver. If the most significant bit is set, the DCB structure is for a parallel device.

BaudRate

Specifies the baud rate at which the communications device operates. If the value of the high-order byte is equal to 0xFF, the low-order byte specifies a baud-rate index. The index can be one of the following values:

CBR_110	CBR_14400
CBR_4400	CBR_19200
CBR_9200	CBR_38400
CBR_8400	CBR_56000
CBR_6000	CBR_128000
CBR_28000	CBR_256000
CBR_9600	

If the high-order byte is not equal to 0xFF, this parameter specifies the actual baud rate.

ByteSize
Specifies the number of bits in the characters transmitted and received. This member can be any number from 4 through 8.

Parity
Specifies the parity scheme to be used. This member can be any one of the following values:

Value	Meaning
EVENPARITY	Even
MARKPARITY	Mark
NOPARITY	No parity
ODDPARITY	Odd

StopBits
Specifies the number of stop bits to be used. This member can be any one of the following values:

Value	Meaning
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 RLSD (receive-line-signal-detect) signal. RLSD is also known as the carrier-detect (CD) signal.

CtsTimeout
Specifies the maximum amount of time, in milliseconds, the device should wait for the CTS (clear-to-send) signal.

DsrTimeout
Specifies the maximum amount of time, in milliseconds, the device should wait for the DSR (data-set-ready) signal.

fBinary
Specifies binary mode. In nonbinary mode, the **EofChar** character is recognized on input and remembered as the end of data.

fRtsDisable
Specifies whether or not the RTS (request-to-send) signal is disabled. If this member is set, RTS is not used and remains low. If this member is clear, RTS is sent when the device is opened and turned off when the device is closed.

fParity
Specifies whether parity checking is enabled. If this member is set, parity checking is performed and errors are reported.

fOutxCtsFlow
Specifies that CTS (clear-to-send) signal is to be monitored for output flow control. If this member is set and CTS is turned off, output is suspended until CTS is again sent.

fOutxDsrFlow
Specifies that the DSR (data-set-ready) signal is to be monitored for output flow control. If this member is set and DSR is turned off, output is suspended until DSR is again sent.

fDummy
Reserved.

fDtrDisable
Specifies whether the DTR (data-terminal-ready) signal is disabled. If this member is set, DTR is not used and remains low. If this member is clear, DTR is sent when the device is opened and turned off when the device is closed.

fOutX
Specifies that XON/XOFF flow control is used during transmission. If this member is set, transmission stops when the **XoffChar** character is received and starts again when the **XonChar** character is received.

fInX
Specifies that XON/XOFF flow control is used during reception. If this member is set, the **XonChar** character is sent when the reception queue comes within **XoffLim** characters of being full and the **XonChar** character is sent when the reception queue comes within **XonLim** characters of being empty.

fPeChar
Specifies that characters received with parity errors are to be replaced with the character specified by this member. This member must be set for the replacement to occur.

fNull
Specifies that received null characters are to be discarded.

fChEvt
Specifies that reception of the **EvtChar** character is to be flagged as an event.

fDtrflow
Specifies that the DTR (data-terminal-ready) signal is to be used for reception flow control. If this member is set, DTR is turned off when the reception queue comes within **XoffLim** characters of being full and sent when the reception queue comes within **XonLim** characters of being empty.

fRtsflow
Specifies that the RTS (ready-to-send) signal is to be used for reception flow control. If this member is set, RTS is turned off when the reception queue comes within **XoffLim** characters of being full, and sent when the reception queue comes within **XonLim** characters of being empty.

fDummy2
Reserved.

XonChar
Specifies the value of the XON character for both transmission and reception.

XoffChar
Specifies the value of the XOFF character for both transmission and reception.

XonLim
Specifies the minimum number of characters allowed in the reception queue before the XON character is sent.

XoffLim
Specifies the maximum number of characters allowed in the reception queue before the XOFF character is sent. The value of the **XoffLim** member is subtracted from the size of the reception queue, in bytes, to calculate the maximum number of characters allowed.

PeChar
Specifies the value of the character used to replace characters received with a parity error.

EofChar
Specifies the value of the character used to signal the end of data.

EvtChar
Specifies the value of the character used to signal an event.

TxDelay
Not currently used.

See Also **BuildCommDCB, GetCommState, SetCommState**

DDEACK

2.x

```
#include <dde.h>

typedef struct tagDDEACK {  /* ddeack */
    WORD bAppReturnCode:8,
         reserved:6,
         fBusy:1,
         fAck:1;
} DDEACK;
```

The **DDEACK** structure contains status flags that a DDE application passes to its partner as part of the WM_DDE_ACK message. The flags provide details about the application's response to a WM_DDE_ADVISE, WM_DDE_DATA, WM_DDE_EXECUTE, WM_DDE_REQUEST, WM_DDE_POKE, or WM_DDE_UNADVISE message.

Members

bAppReturnCode
Specifies an application-defined return code.

fBusy
Indicates whether the application was busy and unable to respond to the partner's message at the time the message was received. A nonzero value indicates the server was busy and unable to respond. The **fBusy** member is defined only when the **fAck** member is zero.

fAck
Indicates whether the application accepted the message from its partner. A nonzero value indicates the server accepted the message.

See Also

WM_DDE_ACK, WM_DDE_ADVISE, WM_DDE_DATA, WM_DDE_EXECUTE, WM_DDE_REQUEST, WM_DDE_POKE, WM_DDE_UNADVISE,

DDEADVISE

2.x

```
#include <dde.h>

typedef struct tagDDEADVISE {  /* ddeadv */
    WORD    reserved:14,
            fDeferUpd:1,
            fAckReq:1;
    short   cfFormat;
} DDEADVISE;
```

The **DDEADVISE** structure contains flags that specify how a server should send data to a client during an advise loop. A client passes the handle of a **DDEADVISE** structure to a server as part of a WM_DDE_ADVISE message.

Members

fDeferUpd
Indicates whether the server should defer sending updated data to the client. A nonzero value tells the server to send a WM_DDE_DATA message with a NULL data handle whenever the data item changes. In response, the client can post a WM_DDE_REQUEST message to the server to obtain a handle to the updated data.

fAckReq
Indicates whether the server should set the **fAckReq** flag in the WM_DDE_DATA messages that it posts to the client. A nonzero value tells the server to set the **fAckReq** bit.

cfFormat
Specifies the client application's preferred data format. The format must be a standard or registered clipboard format. The following standard clipboard formats may be used:

CF_BITMAP	CF_OEMTEXT
CF_DCF_OEMTEXT	CF_PALETTE
CF_DCF_PALETTE	CF_PENDATA
CF_DCF_PENDATA	CF_SYLK
CF_DCF_SYLK	CF_TEXT
CF_DCF_TEXT	CF_TIFF
CF_METAFILEPICT	

See Also

WM_DDE_ADVISE, WM_DDE_DATA, WM_DDE_UNADVISE

DDEDATA

2.x

```
#include <dde.h>

typedef struct tagDDEDATA {    /* ddedat */
    WORD    unused:12,
            fResponse:1,
            fRelease:1,
            reserved:1,
            fAckReq:1;
    short   cfFormat;
    BYTE    Value[1];
} DDEDATA;
```

The **DDEDATA** structure contains the data and information about the data sent as part of a WM_DDE_DATA message.

Members

fResponse
Indicates whether the application receiving the WM_DDE_DATA message should acknowledge receipt of the data by sending a WM_DDE_ACK message. A nonzero value indicates the application should send the acknowledgment.

fRelease
Indicates if the application receiving the WM_DDE_POKE message should free the data. A nonzero value indicates the data should be freed.

fAckReq
Indicates whether the data was sent in response to a WM_DDE_REQUEST message or a WM_DDE_ADVISE message. A nonzero value indicates the data was sent in response to a WM_DDE_REQUEST message.

cfFormat
Specifies the format of the data. The format should be a standard or registered clipboard format. The following standard clipboard formats may be used:

CF_BITMAP	CF_OEMTEXT
CF_DCF_OEMTEXT	CF_PALETTE
CF_DCF_PALETTE	CF_PENDATA
CF_DCF_PENDATA	CF_SYLK
CF_DCF_SYLK	CF_TEXT
CF_DCF_TEXT	CF_TIFF
CF_METAFILEPICT	

See Also

WM_DDE_ACK, WM_DDE_ADVISE, WM_DDE_DATA, WM_DDE_POKE, WM_DDE_REQUEST

DDEPOKE

2.x

```
#include <dde.h>

typedef struct tagDDEPOKE {  /* ddepok */
    WORD  unused:13,
          fRelease:1,
          fReserved:2;
    short cfFormat;
    BYTE  Value[1];
} DDEPOKE;
```

The **DDEPOKE** structure contains the data and information about the data sent as part of a WM_DDE_POKE message.

Members

fRelease
Indicates if the application receiving the WM_DDE_POKE message should free the data. A nonzero value specifies the data should be freed.

cfFormat
Specifies the format of the data. The format should be a standard or registered clipboard format. The following standard clipboard formats may be used:

CF_BITMAP	CF_OEMTEXT
CF_DCF_OEMTEXT	CF_PALETTE
CF_DCF_PALETTE	CF_PENDATA
CF_DCF_PENDATA	CF_SYLK
CF_DCF_SYLK	CF_TEXT
CF_DCF_TEXT	CF_TIFF
CF_METAFILEPICT	

Value
Contains the data. The size of this array depends on the value of the **cfFormat** member.

See Also

WM_DDE_POKE

DEBUGHOOKINFO

3.1

```
typedef struct tagDEBUGHOOKINFO {
    HMODULE hModuleHook;
    LPARAM  reserved;
    LPARAM  lParam;
    WPARAM  wParam;
    int     code;
} DEBUGHOOKINFO;
```

The **DEBUGHOOKINFO** structure contains debugging information.

Members

hModuleHook
Identifies the module containing the filter function.

reserved
Not used.

lParam
Specifies the value to be passed to the hook in the *lParam* parameter of the **DebugProc** callback function.

wParam
Specifies the value to be passed to the hook in the *wParam* parameter of the **DebugProc** callback function.

code
Specifies the value to be passed to the hook in the *code* parameter of the **DebugProc** callback function.

See Also

DebugProc, **SetWindowsHook**

DELETEITEMSTRUCT

3.0

```
typedef struct tagDELETEITEMSTRUCT {    /* deli */
    UINT  CtlType;
    UINT  CtlID;
    UINT  itemID;
    HWND  hwndItem;
    DWORD itemData;
} DELETEITEMSTRUCT;
```

The **DELETEITEMSTRUCT** structure describes a deleted owner-drawn list-box or combo-box item. When an item is removed from the list box or combo box or when the list box or combo box is destroyed, Windows sends the

WM_DELETEITEM message to the owner for each deleted item. The *lParam* parameter of the message contains a pointer to this structure.

Members

CtlType
Contains ODT_LISTBOX (which specifies an owner-drawn list box) or ODT_COMBOBOX (which specifies an owner-drawn combo box).

CtlID
Contains the control identifier for the list box or combo box.

itemID
Contains the index of the item in the list box or combo box being removed.

hwndItem
Contains the window handle of the control.

itemData
Contains the value passed to the control in the *lParam* parameter of the LB_INSERTSTRING, LB_ADDSTRING, CB_INSERTSTRING, or CB_ADDSTRING message when the item was added to the list box.

See Also

WM_DELETEITEM

DEVMODE

3.0

```
#include <print.h>

typedef struct tagDEVMODE {    /* dm */
    char  dmDeviceName[CCHDEVICENAME];
    UINT  dmSpecVersion;
    UINT  dmDriverVersion;
    UINT  dmSize;
    UINT  dmDriverExtra;
    DWORD dmFields;
    int   dmOrientation;
    int   dmPaperSize;
    int   dmPaperLength;
    int   dmPaperWidth;
    int   dmScale;
    int   dmCopies;
    int   dmDefaultSource;
    int   dmPrintQuality;
    int   dmColor;
    int   dmDuplex;
    int   dmYResolution;
    int   dmTTOption;
} DEVMODE;
```

The **DEVMODE** structure contains information about a printer driver's initialization and environment data. An application passes this structure to the **DeviceCapabilities** and **ExtDeviceMode** functions.

Members

dmDeviceName
Specifies the name of the device the driver supports—for example, "PCL/HP LaserJet" in the case of the Hewlett-Packard LaserJet. Each driver has a unique string.

dmSpecVersion
Specifies the version number of the **DEVMODE** structure. For Windows version 3.1, this value should be 0x30A.

dmDriverVersion
Specifies the printer driver version number assigned by the printer driver developer.

dmSize
Specifies the size, in bytes, of the **DEVMODE** structure. (This value does not include the optional **dmDriverData** member for device-specific data, which can follow the structure.) If an application manipulates only the driver-independent portion of the data, it can use this member to find out the length of the structure without having to account for different versions.

dmDriverExtra
Specifies the size, in bytes, of the optional **dmDriverData** member for device-specific data, which can follow the structure. If an application does not use device-specific information, it should set this member to zero.

dmFields
Specifies a set of flags that indicate which of the remaining members in the **DEVMODE** structure have been initialized. It can be any combination (or it can be none) of the following values:

Constant	Value
DM_ORIENTATION	0x0000001L
DM_PAPERSIZE	0x0000002L
DM_PAPERLENGTH	0x0000004L
DM_PAPERWIDTH	0x0000008L
DM_SCALE	0x0000010L
DM_COPIES	0x0000100L
DM_DEFAULTSOURCE	0x0000200L
DM_PRINTQUALITY	0x0000400L
DM_COLOR	0x0000800L
DM_DUPLEX	0x0001000L
DM_YRESOLUTION	0x0002000L
DM_TTOPTION	0x0004000L

A printer driver supports only those members that are appropriate for the printer technology.

dmOrientation
Specifies the orientation of the paper. It can be either DMORIENT_PORTRAIT or DMORIENT_LANDSCAPE.

dmPaperSize
Specifies the size of the paper to print on. This member may be set to zero if the length and width of the paper are specified by the **dmPaperLength** and **dmPaperWidth** members, respectively. Otherwise, the **dmPaperSize** member can be set to one of the following predefined values:

Value	Meaning
DMPAPER_FIRST	DMPAPER_LETTER
DMPAPER_LETTER	Letter, 8 1/2 × 11 in.
DMPAPER_LETTERSMALL	Letter Small, 8 1/2 × 11 in.
DMPAPER_TABLOID	Tabloid, 11 × 17 in.
DMPAPER_LEDGER	Ledger, 17 × 11 in.
DMPAPER_LEGAL	Legal, 8 1/2 × 14 in.
DMPAPER_STATEMENT	Statement, 5 1/2 × 8 1/2 in.
DMPAPER_EXECUTIVE	Executive, 7 1/2 × 10 1/2 in.
DMPAPER_A3	A3, 297 × 420 mm
DMPAPER_A4	A4, 210 × 297 mm
DMPAPER_A4SMALL	A4 Small, 210 × 297 mm
DMPAPER_A5	A5, 148 × 210 mm
DMPAPER_B4	B4, 250 × 354 mm
DMPAPER_B5	B5, 182 × 257 mm
DMPAPER_FOLIO	Folio, 8 1/2 × 13 in.
DMPAPER_QUARTO	Quarto, 215 × 275 mm
DMPAPER_10X14	10 × 14 in.
DMPAPER_11X17	11 × 17 in.
DMPAPER_NOTE	Note, 8 1/2 × 11 in.
DMPAPER_ENV_9	Envelope #9, 3 7/8 × 8 7/8 in.
DMPAPER_ENV_10	Envelope #10, 4 1/8 × 9 1/2 in.
DMPAPER_ENV_11	Envelope #11, 4 1/2 × 10 3/8 in.
DMPAPER_ENV_12	Envelope #12, 4 1/2 × 11 in.
DMPAPER_ENV_14	Envelope #14, 5 × 11 1/2 in.
DMPAPER_CSHEET	C size sheet
DMPAPER_DSHEET	D size sheet
DMPAPER_ESHEET	E size sheet
DMPAPER_ENV_DL	Envelope DL, 110 × 220 mm

Value	Meaning
DMPAPER_ENV_C3	Envelope C3, 324 × 458 mm
DMPAPER_ENV_C4	Envelope C4, 229 × 324 mm
DMPAPER_ENV_C5	Envelope C5, 162 × 229 mm
DMPAPER_ENV_C6	Envelope C6, 114 × 162 mm
DMPAPER_ENV_C65	Envelope C65, 114 × 229 mm
DMPAPER_ENV_B4	Envelope B4, 250 × 353 mm
DMPAPER_ENV_B5	Envelope B5, 176 × 250 mm
DMPAPER_ENV_B6	Envelope B6, 176 × 125 mm
DMPAPER_ENV_ITALY	Envelope, 110 × 230 mm
DMPAPER_ENV_MONARCH	Envelope Monarch, 3 7/8 × 7 1/2 in.
DMPAPER_ENV_PERSONAL	Envelope, 3 5/8 × 6 1/2 in.
DMPAPER_FANFOLD_US	U.S. Standard Fanfold, 14 7/8 × 11 in.
DMPAPER_FANFOLD_STD_GERMAN	German Standard Fanfold, 8 1/2 × 12 in.
DMPAPER_FANFOLD_LGL_GERMAN	German Legal Fanfold, 8 1/2 × 13 in.
DMPAPER_LAST	German Legal Fanfold, 8 1/2 × 13 in.
DMPAPER_USER	User-defined

dmPaperLength
Specifies a paper length, in tenths of a millimeter. This parameter overrides the paper length specified by the **dmPaperSize** member, either for custom paper sizes or for such devices as dot-matrix printers that can print on a variety of page sizes.

dmPaperWidth
Specifies a paper width, in tenths of a millimeter. This parameter overrides the paper width specified by the **dmPaperSize** member.

dmScale
Specifies the factor by which the printed output is to be scaled. The apparent page size is scaled from the physical page size by a factor of **dmScale**/100. For example, a letter-size paper with a **dmScale** value of 50 would contain as much data as a page of size 17 by 22 inches because the output text and graphics would be half their original height and width.

dmCopies
Specifies the number of copies printed if the device supports multiple-page copies.

dmDefaultSource
Specifies the default bin from which the paper is fed. The application can override this value by using the GETSETPAPERBINS escape. This member can be one of the following values:

DMBIN_AUTO
DMBIN_CASSETTE
DMBIN_ENVELOPE
DMBIN_ENVMANUAL
DMBIN_FIRST
DMBIN_LARGECAPACITY
DMBIN_LARGEFMT
DMBIN_LAST
DMBIN_LOWER
DMBIN_MANUAL
DMBIN_MIDDLE
DMBIN_ONLYONE
DMBIN_SMALLFMT
DMBIN_TRACTOR
DMBIN_UPPER

A range of values is reserved for device-specific bins. To be consistent with initialization information, the GETSETPAPERBINS and ENUMPAPERBINS escapes use these values.

dmPrintQuality
Specifies the printer resolution. Following are the four predefined device-independent values:

DMRES_HIGH (–4)
DMRES_MEDIUM (–3)
DMRES_LOW (–2)
DMRES_DRAFT (–1)

If a positive value is given, it specifies the number of dots per inch (DPI) and is therefore device-dependent.

If the printer initializes the **dmYResolution** member, the **dmPrintQuality** member specifies the x-resolution of the printer, in dots per inch.

dmColor
Specifies whether a color printer is to render color or monochrome output. Possible values are:

DMCOLOR_COLOR (1)
DMCOLOR_MONOCHROME (2)

dmDuplex
Specifies duplex (double-sided) printing for printers capable of duplex printing. This member can be one of the following values:

DMDUP_SIMPLEX (1)
DMDUP_HORIZONTAL (2)
DMDUP_VERTICAL (3)

dmYResolution
Specifies the y-resolution of the printer, in dots per inch. If the printer initializes this member, the **dmPrintQuality** member specifies the x-resolution of the printer, in dots per inch.

dmTTOption
Specifies how TrueType fonts should be printed. It can be one of the following values:

Value	Meaning
DMTT_BITMAP	Print TrueType fonts as graphics. This is the default action for dot-matrix printers.
DMTT_DOWNLOAD	Download TrueType fonts as soft fonts. This is the default action for Hewlett-Packard printers that use Printer Control Language (PCL).
DMTT_SUBDEV	Substitute device fonts for TrueType fonts. This is the default action for PostScript printers.

Comments

Only drivers that are fully updated for Windows versions 3.0 and later and that export the **ExtDeviceMode** function use the **DEVMODE** structure.

An application can retrieve the paper sizes and names supported by a printer by calling the **DeviceCapabilities** function with the DC_PAPERS, DC_PAPERSIZE, and DC_PAPERNAMES values.

Before setting the value of the **dmTTOption** member, applications should find out how a printer driver can use TrueType fonts by calling the **DeviceCapabilities** function with the DC_TRUETYPE value.

Drivers can add device-specific data immediately following the **DEVMODE** structure.

See Also

DeviceCapabilities, **ExtDeviceMode**

DEVNAMES

3.1

```
#include <commdlg.h>

typedef struct tagDEVNAMES {    /* dn */
    UINT wDriverOffset;
    UINT wDeviceOffset;
    UINT wOutputOffset;
    UINT wDefault;
    /* optional data may appear here  */
} DEVNAMES;
```

The **DEVNAMES** structure contains offsets to strings that specify the driver, name, and output port of a printer. The **PrintDlg** function uses these strings to initialize controls in the system-defined Print dialog box. When the user chooses the OK button to close the dialog box, information about the selected printer is returned in this structure.

Members

wDriverOffset
Specifies the offset from the beginning of the structure to a null-terminated string that specifies the Microsoft MS-DOS® filename (without extension) of the device driver. On input, this string is used to set which printer to initially display in the dialog box.

wDeviceOffset
Specifies the offset from the beginning of the structure to the null-terminated string that specifies the name of the device. This string cannot exceed 32 bytes in length, including the null character, and must be identical to the **dmDeviceName** member of the **DEVMODE** structure.

wOutputOffset
Specifies the offset from the beginning of the structure to the null-terminated string that specifies the MS-DOS device name for the physical output medium (output port).

wDefault
Specifies whether the strings specified in the **DEVNAMES** structure identify the default printer. It is used to verify that the default printer has not changed since the last print operation. On input, this member can be set to DN_DEFAULTPRN. If the DN_DEFAULTPRN flag is set, the other values in the **DEVNAMES** structure are checked against the current default printer.

On output, the **wDefault** member is changed only if the Print Setup dialog box was displayed and the user chose the OK button to close it. If the default printer was selected, the DN_DEFAULTPRN flag is set. If a printer is specifically selected, the flag is not set. All other bits in this member are reserved for internal use by the dialog box procedure of the Print dialog box.

See Also

PrintDlg

DOCINFO

3.1

```
typedef struct {    /* di */
    int     cbSize;
    LPCSTR  lpszDocName;
    LPCSTR  lpszOutput;
} DOCINFO;
```

The **DOCINFO** structure contains the input and output filenames used by the **StartDoc** function.

Members

cbSize
Specifies the size of the structure, in bytes.

lpszDocName
Points to a null-terminated string specifying the name of the document. This string must not be longer than 32 characters, including the null terminating character.

lpszOutput
Points to a null-terminated string specifying the name of an output file. This allows a print job to be redirected to a file. If this value is NULL, output goes to the device for the specified device context.

See Also

StartDoc

DRAWITEMSTRUCT

3.0

```
typedef struct tagDRAWITEMSTRUCT {  /* ditm */
    UINT  CtlType;
    UINT  CtlID;
    UINT  itemID;
    UINT  itemAction;
    UINT  itemState;
    HWND  hwndItem;
    HDC   hDC;
    RECT  rcItem;
    DWORD itemData;
} DRAWITEMSTRUCT;
```

The **DRAWITEMSTRUCT** structure provides information the owner needs to determine how to paint an owner-drawn control. The owner of the owner-drawn control receives a pointer to this structure as the *lParam* parameter of the WM_DRAWITEM message.

Members

CtlType
Specifies the control type. The values for control types follow:

Value	Meaning
ODT_BUTTON	Owner-drawn button
ODT_COMBOBOX	Owner-drawn combo box
ODT_LISTBOX	Owner-drawn list box
ODT_MENU	Owner-drawn menu

CtlID
Specifies the control identifier for a combo box, list box or button. This member is not used for a menu.

itemID
Specifies the menu-item identifier for a menu or the index of the item in a list box or combo box. For an empty list box or combo box, this member is a negative value. This allows the application to draw only the focus rectangle at the coordinates specified by the **rcItem** member even though there are no items in the control. This indicates to the user whether the list box or combo box has input focus. The **itemAction** member determines whether the rectangle is to be drawn as though the list box or combo box has input focus.

itemAction
Specifies the drawing action required. This member is one or more of the following values:

Value	Meaning
ODA_DRAWENTIRE	Bit is set when the entire control needs to be drawn.
ODA_FOCUS	Bit is set when the control gains or loses input focus. The **itemState** member should be checked to determine whether the control has focus.
ODA_SELECT	Bit is set when only the selection status has changed. The **itemState** member should be checked to determine the new selection state.

itemState
Specifies the visual state of the item after the current drawing action takes place; that is, if a menu item is to be grayed, the state flag ODS_GRAYED will be set. Following are the state flags:

Value	Meaning
ODS_CHECKED	Bit is set if the menu item is to be checked. This bit is used only in a menu.
ODS_DISABLED	Bit is set if the item is to be drawn as disabled.
ODS_FOCUS	Bit is set if the item has input focus.
ODS_GRAYED	Bit is set if the item is to be grayed. This bit is used only in a menu.
ODS_SELECTED	Bit is set if the item's status is selected.

hwndItem
Specifies the window handle of the control for combo boxes, list boxes, and buttons. For menus, it contains the handle of the menu (**HMENU**) containing the item.

hDC
Identifies a device context; this device context must be used when performing drawing operations on the control.

rcItem
Specifies a rectangle in the device context identified by the **hDC** member that defines the boundaries of the control to be drawn. Windows automatically clips anything the owner draws in the device context for combo boxes, list boxes, and buttons, but it does not clip menu items. When drawing menu items, it must ensure that the owner does not draw outside the boundaries of the rectangle defined by the **rcItem** member.

itemData
Contains the value last assigned to the list box or combo box by an LB_SETITEMDATA or CB_SETITEMDATA message. If the list box or combo box has the LBS_HASSTRINGS or CBS_HASSTRINGS style, this value is initially zero. Otherwise, this value is initially the value that was passed to the list box or combo box in the *lParam* parameter of one of the following messages:

CB_ADDSTRING
CB_INSERTSTRING
LB_ADDSTRING
LB_INSERTSTRING

DRIVERINFOSTRUCT

3.1

```
typedef struct tagDRIVERINFOSTRUCT {     /* drvinfst */
    UINT      length;
    HDRVR     hDriver;
    HINSTANCE hModule;
    char      szAliasName[128];
} DRIVERINFOSTRUCT;
```

The **DRIVERINFOSTRUCT** structure contains basic information about an installable device driver.

Members

length
Specifies the size of the **DRIVERINFOSTRUCT** structure.

hDriver
Identifies an instance of the installable driver.

hModule
Identifies an installable driver module.

szAliasName
Points to a null-terminated string that specifies the driver name or an alias under which the driver was loaded.

See Also

GetDriverInfo

DRVCONFIGINFO

3.1

```
typedef struct tagDRVCONFIGINFO {
    DWORD   dwDCISize;
    LPCSTR  lpszDCISectionName;
    LPCSTR  lpszDCIAliasName;
} DRVCONFIGINFO;
```

The **DRVCONFIGINFO** structure contains information about the entries for an installable device driver in the SYSTEM.INI file. This structure is sent in the *lParam* parameter of the DRV_CONFIGURE and DRV_INSTALL installable-driver messages.

Members

dwDCISize
Specifies the size of the **DRVCONFIGINFO** structure.

lpszDCISectionName
Points to a null-terminated string that specifies the name of the section in the SYSTEM.INI file where driver information is recorded.

lpszDCIAliasName
Points to a null-terminated string that specifies the driver name or an alias under which the driver was loaded.

See Also

DRV_CONFIGURE, DRV_INSTALL

EVENTMSG

2.x

```
typedef struct tagEVENTMSG {    /* em */
    UINT  message;
    UINT  paramL;
    UINT  paramH;
    DWORD time;
} EVENTMSG;
```

The **EVENTMSG** structure contains information from the Windows application queue. This structure is used to store message information for the **JournalPlaybackProc** callback function.

Members

message
Specifies the message number.

paramL
Specifies additional information about the message. The exact meaning depends on the **message** value.

paramH
Specifies additional information about the message. The exact meaning depends on the **message** value.

time
Specifies the time at which the message was posted.

See Also **JournalPlaybackProc, SetWindowsHook**

FINDREPLACE

3.1

```
#include <commdlg.h>

typedef struct tagFINDREPLACE {    /* fr */
    DWORD     lStructSize;
    HWND      hwndOwner;
    HINSTANCE hInstance;
    DWORD     Flags;
    LPSTR     lpstrFindWhat;
    LPSTR     lpstrReplaceWith;
    UINT      wFindWhatLen;
    UINT      wReplaceWithLen;
    LPARAM    lCustData;
    UINT      (CALLBACK* lpfnHook)(HWND, UINT, WPARAM, LPARAM);
    LPCSTR    lpTemplateName;
} FINDREPLACE;
```

The **FINDREPLACE** structure contains information that the system uses to initialize a system-defined Find dialog box or Replace dialog box. After the user chooses the OK button to close the dialog box, the system returns information about the user's selections in this structure.

Members

lStructSize
Specifies the length of the structure, in bytes. This member is filled on input.

hwndOwner
Identifies the window that owns the dialog box. This member can be any valid window handle, but it must not be NULL.

If the FR_SHOWHELP flag is set, **hwndOwner** must identify the window that owns the dialog box. The window procedure for this owner window receives a notification message when the user chooses the Help button. (The identifier for the notification message is the value returned by the **RegisterWindowMessage** function when HELPMSGSTRING is passed as its argument.)

This member is filled on input.

hInstance
Identifies a data block that contains a dialog box template specified by the **lpTemplateName** member. This member is only used if the **Flags** member specifies the FR_ENABLETEMPLATE or the FR_ENABLETEMPLATEHANDLE flag; otherwise, this member is ignored. This member is filled on input.

Flags
Specifies the dialog box initialization flags. This member can be a combination of the following values:

Value	Meaning
FR_DIALOGTERM	Indicates the dialog box is closing. The window handle returned by the **FindText** or **ReplaceText** function is no longer valid after this bit is set. This flag is set by the system.
FR_DOWN	Sets the direction of searches through a document. If the flag is set, the search direction is down; if the flag is clear, the search direction is up. Initially, this flag specifies the state of the Up and Down buttons; after the user chooses the OK button to close the dialog box, this flag specifies the user's selection.
FR_ENABLEHOOK	Enables the hook function specified in the **lpfnHook** member of this structure. This flag can be set on input.
FR_ENABLETEMPLATE	Causes the system to use the dialog box template identified by the **hInstance** and **lpTemplateName** members to display the dialog box. This flag is used only to initialize the dialog box.
FR_ENABLETEMPLATEHANDLE	Indicates that the **hInstance** member identifies a data block that contains a preloaded dialog box template. The system ignores the **lpTemplateName** member if this flag is specified. This flag can be set on input.
FR_FINDNEXT	Indicates that the application should search for the next occurrence of the string specified by the **lpstrFindWhat** member. This flag is set by the system.
FR_HIDEMATCHCASE	Hides and disables the Match Case check box. This flag can be set on input.

Value	Meaning
FR_HIDEWHOLEWORD	Hides and disables the Match Only Whole Word check box. This flag can be set on input.
FR_HIDEUPDOWN	Hides the Up and Down radio buttons that control the direction of searches through a document. This flag can be set on input.
FR_MATCHCASE	Specifies that the search is to be case sensitive. This flag is set when the dialog box is created and may be changed by the system in response to user input.
FR_NOMATCHCASE	Disables the Match Case check box. This flag is used only to initialize the dialog box.
FR_NOUPDOWN	Disables the Up and Down buttons. This flag is used only to initialize the dialog box.
FR_NOWHOLEWORD	Disables the Match Whole Word Only check box. This flag is used only to initialize the dialog box.
FR_REPLACE	Indicates that the application should replace the current occurrence of the string specified in the **lpstrFindWhat** member with the string specified in the **lpstrReplaceWith** member. This flag is set by the system.
FR_REPLACEALL	Indicates that the application should replace all occurrences of the string specified in the **lpstrFindWhat** member with the string specified in the **lpstrReplaceWith** member. This flag is set by the system.
FR_SHOWHELP	Causes the dialog box to show the Help button. If this flag is specified, the **hwndOwner** must not be NULL. This flag can be set on input.
FR_WHOLEWORD	Checks the Match Whole Word Only check box. Only whole words that match the search string will be considered. This flag is set when the dialog box is created and may be changed by the system in response to user input.

lpstrFindWhat

Specifies the string to search for. If a string is specified when the dialog box is created, the dialog box will initialize the Find What edit control with this string.

If the FR_FINDNEXT flag is set when the dialog box is created, the application should search for an occurrence of this string (using the FR_DOWN, FR_WHOLEWORD, and FR_MATCHCASE flags to further define the direction and type of search). The application must allocate a buffer for the string. This buffer should be at least 80 bytes long. This flag is set when the dialog box is created and may be changed by the system in response to user input.

lpstrReplaceWith
Specifies the replacement string for replace operations. The **FindText** function ignores this member. The **ReplaceText** function uses this string to initialize the Replace With edit control. This flag is set when the dialog box is created and may be changed by the system in response to user input.

wFindWhatLen
Specifies the length, in bytes, of the buffer to which the **lpstrFindWhat** member points. This member is filled on input.

wReplaceWithLen
Specifies the length, in bytes, of the buffer to which the **lpstrReplaceWith** member points. This member is filled on input.

lCustData
Specifies application-defined data that the system passes to the hook function identified by the **lpfnHook** member. The system passes a pointer to the CHOOSECOLOR structure in the *lParam* parameter of the WM_INITDIALOG message; this pointer can be used to retrieve the **lCustData** member.

lpfnHook
Points to a hook function that processes messages intended for the dialog box. To enable the hook function, an application must specify the FR_ENABLEHOOK flag in the **Flags** member; otherwise, the system ignores this structure member. The hook function must return zero to pass a message that it didn't process back to the dialog box procedure in COMMDLG.DLL. The hook function must return a nonzero value to prevent the dialog box procedure in COMMDLG.DLL from processing a message it has already processed.

This member is filled on input.

lpTemplateName
Points to a null-terminated string that specifies the name of the resource file for the dialog box template that is to be substituted for the dialog box template in COMMDLG.DLL. An application can use the **MAKEINTRESOURCE** macro for numbered dialog box resources. This member is used only if the **Flags** member specifies the FR_ENABLETEMPLATE flag; otherwise, this member is ignored.

This member is filled on input.

Comments

Some members of this structure are filled only when the dialog box is created, some are filled only when the user closes the dialog box, and some have an initialization value that changes when the user closes the dialog box. Whenever a description in the Members section does not specify how the value of a member is assigned, the value is assigned only when the dialog box is created.

See Also

FindText, **ReplaceText**

FIXED

3.1

```
typedef struct tagFIXED {  /* fx */
    UINT    fract;
    int     value;
} FIXED;
```

The **FIXED** structure contains the integral and fractional parts of a fixed-point real number.

Members

fract
Specifies the fractional part of the number.

value
Specifies the integer part of the number.

Comments

The **FIXED** structure is used to describe the elements of the **MAT2** and **POINTFX** structures.

See Also

GetGlyphOutline

FMS_GETDRIVEINFO

```
#include <wfext.h>

typedef struct tagFMS_GETDRIVEINFO { /* fmsgdi */
    DWORD dwTotalSpace;
    DWORD dwFreeSpace;
    char  szPath[260];
    char  szVolume[14];
    char  szShare[128];
} FMS_GETDRIVEINFO, FAR *LPFMS_GETDRIVEINFO;
```

The **FMS_GETDRIVEINFO** structure contains information about the drive that is selected in the currently active File Manager window.

Members

dwTotalSpace
Specifies the total amount of storage space, in bytes, on the disk associated with the drive.

dwFreeSpace
Specifies the amount of free storage space, in bytes, on the disk associated with the drive.

szPath
Specifies a null-terminated string that contains the path of the current directory.

szVolume
Specifies a null-terminated string that contains the volume label of the disk associated with the drive.

szShare
Specifies a null-terminated string that contains the name of the sharepoint (if the drive is being accessed through a network).

See Also

FMExtensionProc, FM_GETDRIVEINFO

FMS_GETFILESEL

```
#include <wfext.h>

typedef struct tagFMS_GETFILESEL { /* fmsgfs */
    UINT  wTime;
    UINT  wDate;
    DWORD dwSize;
    BYTE  bAttr;
    char  szName[260];
} FMS_GETFILESEL;
```

The **FMS_GETFILESEL** structure contains information about a selected file in File Manager's directory window or Search Results window.

Members

wTime
Specifies the time when the file was created.

wDate
Specifies the date when the file was created.

dwSize
Specifies the size, in bytes, of the file.

bAttr
Specifies the attributes of the file.

szName
Specifies a null-terminated string (an OEM string) that contains the fully-qualified path of the selected file. Before displaying this string, an extension should use the **OemToAnsi** function to convert the string to a Windows ANSI string. If a string is to be passed to the MS-DOS file system, an extension should not convert it.

See Also

FMExtensionProc

FMS_LOAD

```
#include <wfext.h>

typedef struct tagFMS_LOAD { /* fmsld */
    DWORD dwSize;
    char  szMenuName[MENU_TEXT_LEN];
    HMENU hMenu;
    UINT  wMenuDelta;
} FMS_LOAD;
```

The **FMS_LOAD** structure contains information that File Manager uses to add a custom menu provided by a File Manager extension dynamic-link library (DLL). The structure also provides a delta value that the extension DLL can use to manipulate the custom menu after File Manager has loaded the menu.

Members

dwSize
Specifies the length of the structure, in bytes.

szMenuName
Contains a null-terminated string for a menu item that appears in File Manager's main menu.

hMenu
Identifies the pop-up menu that is added to File Manager's main menu.

wMenuDelta
Specifies the menu-item delta value. To avoid conflicts with its own menu items, File Manager renumbers the menu-item identifiers in the pop-up menu identified by the **hMenu** member by adding this delta value to each identifier. An extension DLL that needs to modify a menu item must identify the item to modify by adding the delta value to the menu item's identifier. The value of this member can vary from session to session.

See Also

FMExtensionProc

GLOBALENTRY

3.1

```
#include <toolhelp.h>

typedef struct tagGLOBALENTRY {  /* ge */
    DWORD   dwSize;
    DWORD   dwAddress;
    DWORD   dwBlockSize;
    HGLOBAL hBlock;
    WORD    wcLock;
    WORD    wcPageLock;
    WORD    wFlags;
    BOOL    wHeapPresent;
    HGLOBAL hOwner;
    WORD    wType;
    WORD    wData;
    DWORD   dwNext;
    DWORD   dwNextAlt;
} GLOBALENTRY;
```

The **GLOBALENTRY** structure contains information about a memory object on the global heap.

Members

dwSize
Specifies the size of the **GLOBALENTRY** structure, in bytes.

dwAddress
Specifies the linear address of the global-memory object.

dwBlockSize
Specifies the size of the global-memory object, in bytes.

hBlock
Identifies the global-memory object.

wcLock
Specifies the lock count. If this value is zero, the memory object is not locked.

wcPageLock
Specifies the page lock count. If this value is zero, the memory page is not locked.

wFlags
Specifies additional information about the memory object. This member can be the following value:

Value	Meaning
GF_PDB_OWNER	The process data block (PDB) for the task is the owner of the memory object.

wHeapPresent
Indicates whether a local heap exists within the global-memory object.

hOwner
Identifies the owner of the global-memory object.

wType
Specifies the memory type of the object. This type can be one of the following values:

Value	Meaning
GT_UNKNOWN	The memory type is not known.
GT_DGROUP	The object contains the default data segment and the stack segment.
GT_DATA	The object contains program data. (It may also contain stack and local heap data.)
GT_CODE	The object contains program code. If GT_CODE is specified, the **wData** member contains the segment number for the code.
GT_TASK	The object contains the task database.
GT_RESOURCE	The object contains the resource type specified in **wData**.
GT_MODULE	The object contains the module database.
GT_FREE	The object belongs to the free memory pool.
GT_INTERNAL	The object is reserved for internal use by Windows.
GT_SENTINEL	The object is either the first or the last object on the global heap.
GT_BURGERMASTER	The object contains a table that maps selectors to arena handles.

wData
If the **wType** member is not GT_CODE or GT_RESOURCE, **wData** is zero.

If **wType** is GT_CODE, GT_DATA, or GT_DGROUP, **wData** contains the segment number for the code.

If **wType** is GT_RESOURCE, **wData** specifies the type of resource. The type can be one of the following values:

Value	Meaning
GD_ACCELERATORS	The object contains data from the accelerator table.
GD_BITMAP	The object contains data describing a bitmap. This includes the bitmap color table and the bitmap bits.

Value	Meaning
GD_CURSOR	The object contains data describing a group of cursors. This includes the height, width, color count, bit count, and ordinal identifier for the cursors.
GD_CURSORCOMPONENT	The object contains data describing a single cursor. This includes bitmap bits and bitmasks for the cursor.
GD_DIALOG	The object contains data describing controls within a dialog box.
GD_ERRTABLE	The object contains data from the error table.
GD_FONT	The object contains data describing a single font. This data is identical to data in a Windows font file (.FNT).
GD_FONTDIR	The object contains data describing a group of fonts. This includes the number of fonts in the resource and a table of metrics for each of these fonts.
GD_ICON	The object contains data describing a group of icons. This includes the height, width, color count, bit count, and ordinal identifier for the icons.
GD_ICONCOMPONENT	The object contains data describing a single icon. This includes bitmap bits and bitmaps for the icon.
GD_MENU	The object contains menu data for normal and pop-up menu items.
GD_NAMETABLE	The object contains data from the name table.
GD_RCDATA	The object contains data from a user-defined resource.
GD_STRING	The object contains data from the string table.
GD_USERDEFINED	The resource has an unknown resource identifier or is an application-specific named type.

dwNext
Reserved for internal use by Windows.

dwNextAlt
Reserved for internal use by Windows.

See Also

GlobalEntryHandle, GlobalEntryModule, GlobalFirst, GlobalNext, GLOBALINFO

GLOBALINFO

3.1

```
#include <toolhelp.h>

typedef struct tagGLOBALINFO {  /* gi */
    DWORD dwSize;
    WORD  wcItems;
    WORD  wcItemsFree;
    WORD  wcItemsLRU;
} GLOBALINFO;
```

The **GLOBALINFO** structure contains information about the global heap.

Members

dwSize
Specifies the size of the **GLOBALINFO** structure, in bytes.

wcItems
Specifies the total number of items on the global heap.

wcItemsFree
Specifies the number of free items on the global heap.

wcItemsLRU
Specifies the number of "least recently used" (LRU) items on the global heap.

See Also

GlobalInfo, GLOBALENTRY

GLYPHMETRICS

3.1

```
typedef struct tagGLYPHMETRICS {  /* gm */
    UINT  gmBlackBoxX;
    UINT  gmBlackBoxY;
    POINT gmptGlyphOrigin;
    int   gmCellIncX;
    int   gmCellIncY;
} GLYPHMETRICS;
```

The **GLYPHMETRICS** structure contains information about the placement and orientation of a glyph in a character cell.

Members

gmBlackBoxX
Specifies the width of the smallest rectangle that completely encloses the glyph (its "black box").

gmBlackBoxY
Specifies the height of the smallest rectangle that completely encloses the glyph (its "black box").

gmptGlyphOrigin
Specifies the x- and y-coordinates of the upper-left corner of the smallest rectangle that completely encloses the glyph.

gmCellIncX
Specifies the horizontal distance from the origin of the current character cell to the origin of the next character cell.

gmCellIncY
Specifies the vertical distance from the origin of the current character cell to the origin of the next character cell.

Comments

Values in the **GLYPHMETRICS** structure are specified in logical units.

See Also

GetGlyphOutline

HANDLETABLE

2.x

```
typedef struct tagHANDLETABLE {     /* ht */
    HGDIOBJ objectHandle[1];
} HANDLETABLE;
```

The **HANDLETABLE** structure is an array of handles, each of which identifies a graphics device interface (GDI) object.

Members

objectHandle
Contains an array of handles.

See Also

EnumMetaFile, **PlayMetaFileRecord**

HARDWAREHOOKSTRUCT

3.1

```
typedef struct tagHARDWAREHOOKSTRUCT {  /* hhs */
    HWND    hWnd;
    UINT    wMessage;
    WPARAM  wParam;
    LPARAM  lParam;
} HARDWAREHOOKSTRUCT;
```

The **HARDWAREHOOKSTRUCT** contains information about a hardware message placed in the system message queue.

Members

hWnd
Identifies the window that will receive the message.

wMessage
Specifies the message identifier.

wParam
Specifies additional information about the message. The exact meaning depends on the *wMessage* parameter.

lParam
Specifies additional information about the message. The exact meaning depends on the *wMessage* parameter.

HELPWININFO

3.1

```
typedef struct {
    int  wStructSize;
    int  x;
    int  y;
    int  dx;
    int  dy;
    int  wMax;
    char rgchMember[2];
} HELPWININFO;
```

The **HELPWININFO** structure contains the size and position of a secondary help window. An application can set this size by calling the **WinHelp** function with the HELP_SETWINPOS value.

Members

wStructSize
Specifies the size of the **HELPWININFO** structure.

x
Specifies the x-coordinate of the upper-left corner of the window.

y
Specifies the y-coordinate of the upper-left corner of the window.

dx
Specifies the width of the window.

dy
Specifies the height of the window.

wMax
Specifies whether the window should be maximized or set to the given position and dimensions. If this value is 1, the window is maximized. If it is zero, the size and position of the window are determined by the **x**, **y**, **dx**, and **dy** members.

rgchMember
Specifies the name of the window.

Comments

Microsoft Windows Help divides the display into 1024 units in both the x- and y-directions. To create a secondary window that fills the upper-left quadrant of the display, for example, an application would specify zero for the **x** and **y** members and 512 for the **dx** and **dy** members.

See Also

WinHelp

HSZPAIR

3.1

```
#include <ddeml.h>

typedef struct tagHSZPAIR {    /* hp */
    HSZ hszSvc;
    HSZ hszTopic;
} HSZPAIR;
```

The **HSZPAIR** structure contains a dynamic data exchange (DDE) service name and topic name. A DDE server application can use this structure during an XTYP_WILDCONNECT transaction to enumerate the service/topic name pairs that it supports.

Members

hszSvc
Identifies a service name.

hszTopic
Identifies a topic name.

KERNINGPAIR

3.1

```
typedef struct tagKERNINGPAIR {
    WORD wFirst;
    WORD wSecond;
    int  iKernAmount;
} KERNINGPAIR;
```

The **KERNINGPAIR** structure defines a kerning pair.

Members

wFirst
Specifies the character code for the first character in the kerning pair.

wSecond
Specifies the character code for the second character in the kerning pair.

iKernAmount
Specifies the amount that this pair will be kerned if they appear side by side in the same font and size. This value is typically negative, because pair-kerning usually results in two characters being set more tightly than normal. The value is given in logical units—that is, it depends on the current mapping mode.

See Also

GetKerningPairs

LOCALENTRY

3.1

```
#include <toolhelp.h>

typedef struct tagLOCALENTRY {  /* le */
    DWORD   dwSize;
    HLOCAL  hHandle;
    WORD    wAddress;
    WORD    wSize;
    WORD    wFlags;
    WORD    wcLock;
    WORD    wType;
    WORD    hHeap;
    WORD    wHeapType;
    WORD    wNext;
} LOCALENTRY;
```

The **LOCALENTRY** structure contains information about a memory object on the local heap.

Members

dwSize
Specifies the size of the **LOCALENTRY** structure, in bytes.

hHandle
Identifies the local-memory object.

wAddress
Specifies the address of the local-memory object.

wSize
Specifies the size of the local-memory object, in bytes.

wFlags
Specifies whether the memory object is fixed, free, or movable. This member can be one of the following values:

Value	Meaning
LF_FIXED	The object resides in a fixed memory location.
LF_FREE	The object is part of the free memory pool.
LF_MOVEABLE	The object can be moved in order to compact memory.

wcLock
Specifies the lock count. If this value is zero, the memory object is not locked.

wType

Specifies the content of the memory object. This member can be one of the following values:

Value	Meaning
LT_FREE	The object belongs to the free memory pool.
LT_GDI_BITMAP	The object contains a bitmap header.
LT_GDI_BRUSH	The object contains a brush.
LT_GDI_DC	The object contains a device context.
LT_GDI_DISABLED_DC	The object is reserved for internal use by Windows.
LT_GDI_FONT	The object contains a font header.
LT_GDI_MAX	The object is reserved for internal use by Windows.
LT_GDI_METADC	The object contains a metafile device context.
LT_GDI_METAFILE	The object contains a metafile header.
LT_GDI_PALETTE	The object contains a palette.
LT_GDI_PEN	The object contains a pen.
LT_GDI_RGN	The object contains a region.
LT_NORMAL	The object is reserved for internal use by Windows.
LT_USER_ATOMS	The object contains an atom structure.
LT_USER_BWL	The object is reserved for internal use by Windows.
LT_USER_CBOX	The object contains a combo-box structure.
LT_USER_CHECKPOINT	The object is reserved for internal use by Windows.
LT_USER_CLASS	The object contains a class structure.
LT_USER_CLIP	The object is reserved for internal use by Windows.
LT_USER_DCE	The object is reserved for internal use by Windows.
LT_USER_ED	The object contains an edit-control structure.
LT_USER_HANDLETABLE	The object is reserved for internal use by Windows.
LT_USER_HOOKLIST	The object is reserved for internal use by Windows.
LT_USER_HOTKEYLIST	The object is reserved for internal use by Windows.
LT_USER_LBIV	The object contains a list-box structure.
LT_USER_LOCKINPUTSTATE	The object is reserved for internal use by Windows.
LT_USER_MENU	The object contains a menu structure.
LT_USER_MISC	The object is reserved for internal use by Windows.
LT_USER_MWP	The object is reserved for internal use by Windows.
LT_USER_OWNERDRAW	The object is reserved for internal use by Windows.
LT_USER_PALETTE	The object is reserved for internal use by Windows.
LT_USER_POPUPMENU	The object is reserved for internal use by Windows.
LT_USER_PROP	The object contains a window-property structure.

Value	Meaning
LT_USER_SPB	The object is reserved for internal use by Windows.
LT_USER_STRING	The object is reserved for internal use by Windows.
LT_USER_USERSEEUSERDOALLOC	The object is reserved for internal use by Windows.
LT_USER_WND	The object contains a window structure.

hHeap
Identifies the local-memory heap.

wHeapType
Specifies the type of local heap. This type can be one of the following values:

Value	Meaning
NORMAL_HEAP	The heap is the default heap.
USER_HEAP	The heap is used by the USER module.
GDI_HEAP	The heap is used by the GDI module.

wNext
Specifies the next entry in the local heap. This member is reserved for internal use by Windows.

Comments

The **wType** values are for informational purposes only. Microsoft reserves the right to change or delete these tags at any time. Applications should never directly change items on the system heaps, as this information will change in future versions. The **wType** values for the USER module are included only in the debugging versions of USER.EXE.

See Also

LocalFirst, LocalNext, LOCALINFO

LOCALINFO

3.1

```
#include <toolhelp.h>

typedef struct tagLOCALINFO {  /* li */
    DWORD  dwSize;
    WORD   wcItems;
} LOCALINFO;
```

The **LOCALINFO** structure contains information about the local heap.

Members

dwSize
Specifies the size of the **LOCALINFO** structure, in bytes.

wcItems
Specifies the total number of items on the local heap.

See Also

LocalInfo, LOCALENTRY

LOGBRUSH

2.x

```
typedef struct tagLOGBRUSH {    /* lb */
    UINT     lbStyle;
    COLORREF lbColor;
    int      lbHatch;
} LOGBRUSH;
```

The **LOGBRUSH** structure defines the style, color, and pattern of a physical brush to be created by using the **CreateBrushIndirect** function.

Members

lbStyle
Specifies the brush style. This member can be one of the following values:

Value	Meaning
BS_DIBPATTERN	Specifies a pattern brush defined by a device-independent bitmap (DIB) specification.
BS_HATCHED	Specifies a hatched brush.
BS_HOLLOW	Specifies a hollow brush.
BS_PATTERN	Specifies a pattern brush defined by a memory bitmap.
BS_NULL	Equivalent to BS_HOLLOW.
BS_SOLID	Specifies a solid brush.

lbColor

Specifies the color in which the brush is to be drawn. If the **lbStyle** member is the BS_HOLLOW or BS_PATTERN value, **lbColor** is ignored.

If **lpStyle** is the BS_DIBPATTERN value, the low-order word of **lbColor** specifies whether the **bmiColors** members of the **BITMAPINFO** structure contain explicit RGB values or indexes into the currently realized logical palette. The **lbColor** member must be one of the following values:

Value	Meaning
DIB_PAL_COLORS	Color table consists of an array of 16-bit indexes into the currently realized logical palette.
DIB_RGB_COLORS	Color table contains literal RGB values.

lbHatch

Specifies a hatch style. The meaning depends on the brush style.

If the **lbStyle** member is the BS_DIBPATTERN style, the **lbHatch** member contains a handle to a packed DIB. To obtain this handle, an application calls the **GlobalAlloc** function to allocate a global memory object and then fills the memory with the packed DIB. A packed DIB consists of a **BITMAPINFO** structure immediately followed by the array of bytes which define the pixels of the bitmap.

If the **lbStyle** member is the BS_HATCHED style, the **lbHatch** member specifies the orientation of the lines used to create the hatch. This member can be one of the following values:

Value	Meaning
HS_BDIAGONAL	45-degree upward hatch (left to right)
HS_CROSS	Horizontal and vertical cross-hatch
HS_DIAGCROSS	45-degree cross-hatch
HS_FDIAGONAL	45-degree downward hatch (left to right)
HS_HORIZONTAL	Horizontal hatch
HS_VERTICAL	Vertical hatch

If the **lbStyle** member is the BS_PATTERN style, **lbHatch** must be a handle to the bitmap that defines the pattern.

If the **lbStyle** member is the BS_SOLID or the BS_HOLLOW style, **lbHatch** is ignored.

See Also **BITMAPINFO, CreateBrushIndirect, CreateBrushIndirect, GlobalAlloc**

LOGFONT

2.x

```
typedef struct tagLOGFONT {     /* lf */
    int   lfHeight;
    int   lfWidth;
    int   lfEscapement;
    int   lfOrientation;
    int   lfWeight;
    BYTE  lfItalic;
    BYTE  lfUnderline;
    BYTE  lfStrikeOut;
    BYTE  lfCharSet;
    BYTE  lfOutPrecision;
    BYTE  lfClipPrecision;
    BYTE  lfQuality;
    BYTE  lfPitchAndFamily;
    BYTE  lfFaceName[LF_FACESIZE];
} LOGFONT;
```

The **LOGFONT** structure defines the attributes of a font, a drawing object used to write text on a display surface.

Members

lfHeight

Specifies the desired height, in logical units, for the font. If this value is greater than zero, it specifies the cell height of the font. If it is less than zero, it specifies the character height of the font. (Character height is the cell height minus the internal leading. Applications that specify font height in points typically use a negative number for this member.) If this value is zero, the font mapper uses a default height. The font mapper chooses the largest physical font that does not exceed the requested size (or the smallest font, if all the fonts exceed the requested size). The absolute value of the **lfHeight** member must not exceed 16,384 after it is converted to device units.

lfWidth

Specifies the average width, in logical units, of characters in the font. If this value is zero, the font mapper chooses a reasonable default width for the specified font height. (The default width is chosen by matching the aspect ratio of the device against the digitization aspect ratio of the available fonts. The closest match is determined by the absolute value of the difference.) The widths of characters in TrueType fonts are scaled by a factor of this member divided by the width of the characters in the physical font (as specified by the **tmAveCharWidth** member of the **TEXTMETRIC** structure).

lfEscapement

Specifies the angle, in tenths of degrees, between the base line of a character and the x-axis. The angle is measured in a counterclockwise direction from the x-axis for left-handed coordinate systems (that is, MM_TEXT, in which the y direction is down) and in a clockwise direction from the x-axis for right-handed coordinate systems (in which the y direction is up).

lfOrientation
Specifies the orientation of the characters. This value is ignored.

lfWeight
Specifies the font weight. This member can be one of the following values:

Constant	Value
FW_DONTCARE	0
FW_THIN	100
FW_EXTRALIGHT	200
FW_ULTRALIGHT	200
FW_LIGHT	300
FW_NORMAL	400
FW_REGULAR	400
FW_MEDIUM	500
FW_SEMIBOLD	600
FW_DEMIBOLD	600
FW_BOLD	700
FW_EXTRABOLD	800
FW_ULTRABOLD	800
FW_BLACK	900
FW_HEAVY	900

The actual appearance of the font depends on the type face. Some fonts have only FW_NORMAL, FW_REGULAR, and FW_BOLD weights. If FW_DONTCARE is specified, a default weight is used.

lfItalic
Specifies an italic font if nonzero.

lfUnderline
Specifies an underlined font if nonzero.

lfStrikeOut
Specifies a strikeout font if nonzero.

lfCharSet
Specifies the character set of the font. The following values are predefined:

Constant	Value
ANSI_CHARSET	0
DEFAULT_CHARSET	1
SYMBOL_CHARSET	2
SHIFTJIS_CHARSET	128
OEM_CHARSET	255

The DEFAULT_CHARSET value is not used by the font mapper. An application can use this value to allow the name and size of a font to fully describe the logical font. If the specified font name does not exist, a font from any character set can be substituted for the specified font; applications should use the DEFAULT_CHARSET value sparingly to avoid unexpected results.

The OEM character set is system-dependent.

Fonts with other character sets may exist in the system. If an application uses a font with an unknown character set, it should not attempt to translate or interpret strings that are to be rendered with that font.

lfOutPrecision

Specifies the desired output precision. The output precision defines how closely the output must match the height, width, character orientation, escapement, and pitch of the requested font. This member can be one of the following values:

OUT_CHARACTER_PRECIS	OUT_STRING_PRECIS
OUT_DEFAULT_PRECIS	OUT_STROKE_PRECIS
OUT_DEVICE_PRECIS	OUT_TT_PRECIS
OUT_RASTER_PRECIS	OUT_TT_ONLY_PRECIS

Applications can use the values OUT_DEVICE_PRECIS, OUT_RASTER_PRECIS, and OUT_TT_PRECIS to control how the font mapper chooses a font when the system contains more than one font with a given name. For example, if a system contains a font named "Symbol" in raster and TrueType form, specifying OUT_TT_PRECIS would force the font mapper to choose the TrueType version. (Specifying OUT_TT_PRECIS forces the font mapper to choose a TrueType font whenever the specified font name matches a device or raster font, even when there is no TrueType font with the same name.)

An application can use TrueType fonts exclusively by specifying OUT_TT_ONLY_PRECIS. When this value is specified, the system chooses a TrueType font even when the name specified in the **lfFaceName** member matches a raster or vector font.

lfClipPrecision

Specifies the desired clipping precision. The clipping precision defines how to clip characters that are partially outside the clipping region. This member can be any one of the following values:

CLIP_CHARACTER_PRECIS	CLIP_MASK
CLIP_DEFAULT_PRECIS	CLIP_STROKE_PRECIS
CLIP_EMBEDDED	CLIP_TT_ALWAYS
CLIP_LH_ANGLES	

To use an embedded read-only font, applications must specify the CLIP_EMBEDDED value.

To achieve consistent rotation of device, TrueType, and vector fonts, an application can use the OR operator to combine the CLIP_LH_ANGLES value with

any of the other **lfClipPrecision** values. If the CLIP_LH_ANGLES bit is set, the rotation for all fonts is dependent on whether the orientation of the coordinate system is left-handed or right-handed. If CLIP_LH_ANGLES is not set, device fonts always rotate counter-clockwise, but the rotation of other fonts is dependent on the orientation of the coordinate system. (For more information about the orientation of coordinate systems, see the description of the **lfEscapement** member.)

lfQuality

Specifies the output quality of the font, which defines how carefully the graphics device interface (GDI) must attempt to match the logical-font attributes to those of an actual physical font. This member can be one of the following values:

Value	Meaning
DEFAULT_QUALITY	Appearance of the font does not matter.
DRAFT_QUALITY	Appearance of the font is less important than when the PROOF_QUALITY value is used. For GDI raster fonts, scaling is enabled. Bold, italic, underline, and strikeout fonts are synthesized if necessary.
PROOF_QUALITY	Character quality of the font is more important than exact matching of the logical-font attributes. For GDI raster fonts, scaling is disabled and the font closest in size is chosen. Bold, italic, underline, and strikeout fonts are synthesized if necessary.

lfPitchAndFamily

Specifies the pitch and family of the font. The two low-order bits, which specify the pitch of the font, can be one of the following values:

DEFAULT_PITCH
FIXED_PITCH
VARIABLE_PITCH

The four high-order bits of the member, which specify the font family, can be one of the following values:

Value	Meaning
FF_DECORATIVE	Novelty fonts. Old English is an example.
FF_DONTCARE	Don't care or don't know.
FF_MODERN	Fonts with constant stroke width, with or without serifs. Pica, Elite, and Courier New® are examples.
FF_ROMAN	Fonts with variable stroke width and with serifs. Times New Roman® and New Century Schoolbook® are examples.

Value	Meaning
FF_SCRIPT	Fonts designed to look like handwriting. Script and Cursive are examples.
FF_SWISS	Fonts with variable stroke width and without serifs. MS® Sans Serif is an example.

An application can specify a value for the **lfPitchAndFamily** member by using the Boolean OR operator to join a pitch constant with a family constant.

Font families describe the look of a font in a general way. They are intended for specifying fonts when the exact typeface desired is not available.

lfFaceName
Specifies the typeface name of the font. The length of this string must not exceed LF_FACESIZE – 1. The **EnumFontFamilies** function can be used to enumerate the typeface names of all currently available fonts. If **lfFaceName** is NULL, GDI uses a device-dependent typeface.

Comments

Applications can use the default settings for most of these members when creating a logical font. The members that should always be given specific values are **lfHeight** and **lfFaceName**. If **lfHeight** and **lfFaceName** are not set by the application, the logical font that is created is device-dependent.

See Also

CreateFontIndirect, **EnumFontFamilies**

LOGPALETTE

3.0

```
typedef struct tagLOGPALETTE {  /* lgpl */
    WORD         palVersion;
    WORD         palNumEntries;
    PALETTEENTRY palPalEntry[1];
} LOGPALETTE;
```

The **LOGPALETTE** structure defines a logical color palette.

Members

palVersion
Specifies the Windows version number for the structure. This value should be 0x300 for Windows 3.0 and later.

palNumEntries
Specifies the number of palette color entries.

palPalEntry
Specifies an array of **PALETTEENTRY** structures that define the color and usage of each entry in the logical palette.

Comments

The colors in the palette entry table should appear in order of importance, because entries earlier in the logical palette are most likely to be placed in the system palette.

This structure is passed as a parameter to the **CreatePalette** function.

See Also

CreatePalette, **PALETTEENTRY**

LOGPEN

2.x

```
typedef struct tagLOGPEN {  /* lgpn */
    UINT     lopnStyle;
    POINT    lopnWidth;
    COLORREF lopnColor;
} LOGPEN;
```

The **LOGPEN** structure defines the style, width, and color of a pen, a drawing object used to draw lines and borders. The **CreatePenIndirect** function uses the **LOGPEN** structure.

Members

lopnStyle

Specifies the pen type. This member can be one of the following values:

Value	Meaning
PS_SOLID	Creates a solid pen.
PS_DASH	Creates a dashed pen. (Valid only when the pen width is 1.)
PS_DOT	Creates a dotted pen. (Valid only when the pen width is 1.)
PS_DASHDOT	Creates a pen with alternating dashes and dots. (Valid only when the pen width is 1.)
PS_DASHDOTDOT	Creates a pen with alternating dashes and double dots. (Valid only when the pen width is 1.)
PS_NULL	Creates a null pen.
PS_INSIDEFRAME	Creates a pen that draws a line inside the frame of closed shapes produced by graphics device interface (GDI) output functions that specify a bounding rectangle (for example, the **Ellipse**, **Rectangle**, **RoundRect**, **Pie**, and **Chord** functions). When this style is used with GDI output functions that do not specify a bounding rectangle (for example, the **LineTo** function), the drawing area of the pen is not limited by a frame.

If a pen has the PS_INSIDEFRAME style and a color that does not match a color in the logical color table, the pen is drawn with a dithered color. The PS_SOLID pen style cannot be used to create a pen with a dithered color. The PS_INSIDEFRAME style is identical to PS_SOLID if the pen width is less than or equal to 1.

When the PS_INSIDEFRAME style is used with GDI objects produced by functions other than **Ellipse**, **Rectangle**, and **RoundRect**, the line may not be completely inside the specified frame.

lopnWidth
Specifies the pen width, in logical units. If the **lopnWidth** member is zero, the pen is one pixel wide on raster devices regardless of the current mapping mode.

lopnColor
Specifies the pen color.

Comments

The *y* value in the **POINT** structure for the **lopnWidth** member is not used.

See Also

CreatePenIndirect, **POINT**

MAT2

3.1

```
typedef struct tagMAT2 {  /* mat2 */
    FIXED eM11;
    FIXED eM12;
    FIXED eM21;
    FIXED eM22;
} MAT2;
```

The **MAT2** structure contains the values for a transformation matrix.

Members

eM11
Specifies a fixed-point value for the *M11* component of a 2-by-2 transformation matrix.

eM12
Specifies a fixed-point value for the *M12* component of a 2-by-2 transformation matrix.

eM21
Specifies a fixed-point value for the *M21* component of a 2-by-2 transformation matrix.

eM22
Specifies a fixed-point value for the *M22* component of a 2-by-2 transformation matrix.

Comments

The identity matrix produces a transformation in which the transformed graphical object is identical to the source object. In the identity matrix, the value of **eM11** is 1, the value of **eM12** is zero, the value of **eM21** is zero, and the value of **eM22** is 1.

See Also

GetGlyphOutline

MDICREATESTRUCT

3.0

```
typedef struct tagMDICREATESTRUCT {     /* mdic */
    LPCSTR    szClass;
    LPCSTR    szTitle;
    HINSTANCE hOwner;
    int       x;
    int       y;
    int       cx;
    int       cy;
    DWORD     style;
    LPARAM    lParam;
} MDICREATESTRUCT;
```

The **MDICREATESTRUCT** structure contains information about the class, title, owner, location, and size of a multiple document interface (MDI) child window.

Members

szClass
Contains a long pointer to the application-defined class of the MDI child window.

szTitle
Contains a long pointer to the window title of the MDI child window.

hOwner
Identifies the instance handle of the application creating the MDI child window.

x
Specifies the initial position of the left side of the MDI child window. If this member is set to CW_USEDEFAULT, the MDI child window is assigned a default horizontal position.

y
Specifies the initial position of the top edge of the MDI child window. If this member is set to CW_USEDEFAULT, the MDI child window is assigned a default vertical position.

cx
Specifies the initial width of the MDI child window. If this member is set to CW_USEDEFAULT, the MDI child window is assigned a default width.

cy
Specifies the initial height of the MDI child window. If this member is set to CW_USEDEFAULT, the MDI child window is assigned a default height.

style
Specifies additional styles for the MDI child window. If the window was created with the MDIS_ALLCHILDSTYLES window style, the **style** member may be any combination of the window styles documented with the **CreateWindow** function. Otherwise, it may be one or more of the following values:

Value	Meaning
WS_MINIMIZE	MDI child window is created in a minimized state.
WS_MAXIMIZE	MDI child window is created in a maximized state.
WS_HSCROLL	MDI child window is created with a horizontal scroll bar.
WS_VSCROLL	MDI child window is created with a vertical scroll bar.

lParam
Specifies an application-defined 32-bit value.

Comments

When the MDI child window is created, Windows sends the WM_CREATE message to the window. The *lParam* parameter of the WM_CREATE message contains a pointer to a **CREATESTRUCT** structure. The **lpCreateParams** member of **CREATESTRUCT** contains a pointer to the **MDICREATESTRUCT** structure passed with the WM_MDICREATE message that created the MDI child window.

See Also

CREATESTRUCT

MEASUREITEMSTRUCT

3.0

```
typedef struct tagMEASUREITEMSTRUCT {    /* mi */
    UINT    CtlType;
    UINT    CtlID;
    UINT    itemID;
    UINT    itemWidth;
    UINT    itemHeight;
    DWORD   itemData;
} MEASUREITEMSTRUCT;
```

The **MEASUREITEMSTRUCT** structure informs Windows of the dimensions of an owner-drawn control. This allows Windows to process user interaction with the control correctly. The owner of an owner-drawn control receives a pointer to this structure as the *lParam* parameter of an WM_MEASUREITEM message. The owner-drawn control sends this message to its owner window when the control is created. The owner then fills in the appropriate members in the structure for the control and returns. This structure is common to all owner-drawn controls.

Members

CtlType

Specifies the control type. The values for control types are as follows:

Value	Meaning
ODT_BUTTON	Owner-drawn button
ODT_COMBOBOX	Owner-drawn combo box
ODT_LISTBOX	Owner-drawn list box
ODT_MENU	Owner-drawn menu

CtlID

Specifies the control identifier for a combo box, list box, or button. This member is not used for a menu.

itemID

Specifies the menu-item identifier for a menu or the list-box item identifier for a variable-height combo box or list box. This member is not used for a fixed-height combo box or list box or for a button.

itemWidth

Specifies the width of a menu item. The owner of the owner-drawn menu item must fill this member before returning from the message.

itemHeight

Specifies the height of an individual item in a list box or a menu. Before returning from the message, the owner of the owner-drawn combo box, list box, or menu item must fill out this member. The maximum height of a list box item is 255.

itemData
Contains the value that was passed to the combo box or list box in the *lParam* parameter of one of the following messages:

CB_ADDSTRING
CB_INSERTSTRING
LB_ADDSTRING
LB_INSERTSTRING

Comments

Failure to fill out the proper members in the **MEASUREITEMSTRUCT** structure will cause improper operation of the control.

MEMMANINFO

3.1

```
#include <toolhelp.h>

typedef struct tagMEMMANINFO {  /* mmi */
    DWORD dwSize;
    DWORD dwLargestFreeBlock;
    DWORD dwMaxPagesAvailable;
    DWORD dwMaxPagesLockable;
    DWORD dwTotalLinearSpace;
    DWORD dwTotalUnlockedPages;
    DWORD dwFreePages;
    DWORD dwTotalPages;
    DWORD dwFreeLinearSpace;
    DWORD dwSwapFilePages;
    WORD  wPageSize;
} MEMMANINFO;
```

The **MEMMANINFO** structure contains information about the status and performance of the virtual-memory manager. If the memory manager is running in standard mode, the only valid member of this structure is the **dwLargestFreeBlock** member.

Members

dwSize
Specifies the size of the **MEMMANINFO** structure, in bytes.

dwLargestFreeBlock
Specifies the largest free block of contiguous linear memory in the system, in bytes.

dwMaxPagesAvailable
Specifies the maximum number of pages that could be allocated in the system (the value of **dwLargestFreeBlock** divided by the value of **wPageSize**).

dwMaxPagesLockable
Specifies the maximum number of pages that could be allocated and locked.

dwTotalLinearSpace
Specifies the size of the total linear address space, in pages.

dwTotalUnlockedPages
Specifies the number of unlocked pages in the system. This value includes free pages.

dwFreePages
Specifies the number of pages that are not in use.

dwTotalPages
Specifies the total number of pages the virtual-memory manager manages. This value includes free, locked, and unlocked pages.

dwFreeLinearSpace
Specifies the amount of free memory in the linear address space, in pages.

dwSwapFilePages
Specifies the number of pages in the system swap file.

wPageSize
Specifies the system page size, in bytes.

See Also **MemManInfo**

MENUITEMTEMPLATE

3.0

```
typedef  struct {   /* mit */
    UINT mtOption;
    UINT mtID;
    char mtString[1];
} MENUITEMTEMPLATE;
```

The **MENUITEMTEMPLATE** structure defines a menu item.

Members

mtOption
Specifies a mask of one or more predefined menu options that specify the appearance of the menu item. The menu options follow:

Value	Meaning
MF_CHECKED	Item has a check mark next to it.
MF_GRAYED	Item is initially inactive and drawn with a gray effect.
MF_HELP	Item has a vertical separator to its left.

Value	Meaning
MF_MENUBARBREAK	Item is placed in a new column. The old and new columns are separated by a bar.
MF_MENUBREAK	Item is placed in a new column.
MF_OWNERDRAW	Owner of the menu is responsible for drawing all visual aspects of the menu item, including highlighted, checked and inactive states. This option is not valid for a top-level menu item.
MF_POPUP	Item displays a sublist of menu items when selected.

mtID
Specifies an identification code for a non-pop-up menu item. The **MENUITEMTEMPLATE** structure for a pop-up menu item does not contain the **mtID** member.

mtString
Specifies a null-terminated string that contains the name of the menu item.

See Also **LoadMenuIndirect, MENUITEMTEMPLATEHEADER**

MENUITEMTEMPLATEHEADER

3.0

```
typedef struct {    /* mith */
    UINT    versionNumber;
    UINT    offset;
} MENUITEMTEMPLATEHEADER;
```

A complete menu template consists of a header and one or more menu-item lists.

Members

versionNumber
Specifies the version number. This member should be zero.

offset
Specifies the offset from the end of the header, in bytes, where the menu-item list begins.

Comments One or more **MENUITEMTEMPLATE** structures are combined to form the menu-item list.

See Also **MENUITEMTEMPLATE**

METAFILEPICT

2.x

```
typedef struct tagMETAFILEPICT {    /* mfp */
   int       mm;
   int       xExt;
   int       yExt;
   HMETAFILE hMF;
} METAFILEPICT;
```

The **METAFILEPICT** structure defines the metafile picture format used for exchanging metafile data through the clipboard.

Members

mm
Specifies the mapping mode in which the picture is drawn.

xExt
Specifies the size of the metafile picture for all modes except the MM_ISOTROPIC and MM_ANISOTROPIC modes. The x-extent specifies the width of the rectangle within which the picture is drawn. The coordinates are in units that correspond to the mapping mode.

yExt
Specifies the size of the metafile picture for all modes except the MM_ISOTROPIC and MM_ANISOTROPIC modes. The y-extent specifies the height of the rectangle within which the picture is drawn. The coordinates are in units that correspond to the mapping mode.

For MM_ISOTROPIC and MM_ANISOTROPIC modes, which can be scaled, the **xExt** and **yExt** members contain an optional suggested size in MM_HIMETRIC units. For MM_ANISOTROPIC pictures, **xExt** and **yExt** can be zero when no suggested size is supplied. For MM_ISOTROPIC pictures, an aspect ratio must be supplied even when no suggested size is given. (If a suggested size is given, the aspect ratio is implied by the size.) To give an aspect ratio without implying a suggested size, set **xExt** and **yExt** to negative values whose ratio is the appropriate aspect ratio. The magnitude of the negative **xExt** and **yExt** values will be ignored; only the ratio will be used.

hMF
Identifies a memory metafile.

See Also

SetClipboardData

METAHEADER

3.1

```
typedef struct tagMETAHEADER {  /* mh */
    UINT  mtType;
    UINT  mtHeaderSize;
    UINT  mtVersion;
    DWORD mtSize;
    UINT  mtNoObjects;
    DWORD mtMaxRecord;
    UINT  mtNoParameters;
} METAHEADER;
```

The **METAHEADER** structure contains information about a metafile.

Members

mtType
Specifies whether the metafile is in memory or recorded in a disk file. This member can be one of the following values:

Value	Meaning
1	Metafile is in memory.
2	Metafile is in a disk file.

mtHeaderSize
Specifies the size, in words, of the metafile header.

mtVersion
Specifies the Windows version number. The version number for metafiles that support device-independent bitmaps (DIBs) is 0x0300. Otherwise, the version number is 0x0100.

mtSize
Specifies the size, in words, of the file.

mtNoObjects
Specifies the maximum number of objects that exist in the metafile at the same time.

mtMaxRecord
Specifies the size, in words, of the largest record in the metafile.

mtNoParameters
Reserved.

See Also

METARECORD

METARECORD

3.1

```
typedef struct tagMETARECORD {  /* mr */
    DWORD rdSize;
    UINT  rdFunction;
    UINT  rdParm[1];
} METARECORD;
```

The **METARECORD** structure contains a metafile record.

Members

rdSize
Specifies the size, in words, of the record.

rdFunction
Specifies the function number.

rdParm
Specifies an array of words containing the function parameters, in the reverse order in which they are passed to the function.

See Also

METAHEADER

MINMAXINFO

3.1

```
typedef struct tagMINMAXINFO {  /* mmi */
    POINT ptReserved;
    POINT ptMaxSize;
    POINT ptMaxPosition;
    POINT ptMinTrackSize;
    POINT ptMaxTrackSize;
} MINMAXINFO;
```

The **MINMAXINFO** structure contains information about a window's maximized size and position and its minimum and maximum tracking size.

Members

ptReserved
Reserved for internal use.

ptMaxSize
Specifies the maximized width (*point*.**x**) and the maximized height (*point*.**y**) of the window.

ptMaxPosition
Specifies the position of the left side of the maximized window (*point*.**x**) and the position of the top of the maximized window (*point*.**y**).

ptMinTrackSize
Specifies the minimum tracking width (*point*.**x**) and the minimum tracking height (*point*.**y**) of the window.

ptMaxTrackSize
Specifies the maximum tracking width (*point*.**x**) and the maximum tracking height (*point*.**y**) of the window.

See Also **POINT**, WM_GETMINMAXINFO

MODULEENTRY

3.1

```
#include <toolhelp.h>

typedef struct tagMODULEENTRY {  /* me */
    DWORD   dwSize;
    char    szModule[MAX_MODULE_NAME + 1];
    HMODULE hModule;
    WORD    wcUsage;
    char    szExePath[MAX_PATH + 1];
    WORD    wNext;
} MODULEENTRY;
```

The **MODULEENTRY** structure contains information about one module in the module list.

Members

dwSize
Specifies the size of the **MODULEENTRY** structure, in bytes.

szModule
Specifies the null-terminated string that contains the module name.

hModule
Identifies the module handle.

wcUsage
Specifies the reference count of the module. This is the same number returned by the **GetModuleUsage** function.

szExePath
Specifies the null-terminated string that contains the fully-qualified executable path for the module.

wNext
Specifies the next module in the module list. This member is reserved for internal use by Windows.

See Also **ModuleFindHandle, ModuleFindName, ModuleFirst, ModuleNext**

MONCBSTRUCT

3.1

```
#include <ddeml.h>

typedef struct tagMONCBSTRUCT {    /* mcbst */
    UINT     cb;
    WORD     wReserved;
    DWORD    dwTime;
    HANDLE   hTask;
    DWORD    dwRet;
    UINT     wType;
    UINT     wFmt;
    HCONV    hConv;
    HSZ      hsz1;
    HSZ      hsz2;
    HDDEDATA hData;
    DWORD    dwData1;
    DWORD    dwData2;
} MONCBSTRUCT;
```

The **MONCBSTRUCT** structure contains information about the current dynamic data exchange (DDE) transaction. A DDE debugging application can use this structure when monitoring transactions that the system passes to the DDE callback functions of other applications.

Members

cb
Specifies the length, in bytes, of the structure.

wReserved
Reserved.

dwTime
Specifies the Windows time at which the transaction occurred. Windows time is the number of milliseconds that have elapsed since the system was started.

hTask
Identifies the task (application instance) containing the DDE callback function that received the transaction.

dwRet
Specifies the value returned by the DDE callback function that processed the transaction.

wType
Specifies the transaction type.

wFmt
Specifies the format of the data (if any) exchanged during the transaction.

hConv
Identifies the conversation in which the transaction took place.

hsz1
Identifies a string.

hsz2
Identifies a string.

hData
Identifies the data (if any) exchanged during the transaction.

dwData1
Specifies additional data.

dwData2
Specifies additional data.

See Also

MONERRSTRUCT, MONHSZSTRUCT, MONLINKSTRUCT, MONMSGSTRUCT

MONCONVSTRUCT

3.1

```
#include <ddeml.h>

typedef struct tagMONCONVSTRUCT { /* mcvst */
    UINT    cb;
    BOOL    fConnect;
    DWORD   dwTime;
    HANDLE  hTask;
    HSZ     hszSvc;
    HSZ     hszTopic;
    HCONV   hConvClient;
    HCONV   hConvServer;
} MONCONVSTRUCT;
```

The **MONCONVSTRUCT** structure contains information about a conversation. A dynamic data exchange (DDE) monitoring application can use this structure to obtain information about an advise loop that has been established or terminated.

Members

cb
Specifies the length, in bytes, of the structure.

fConnect
Indicates whether the conversation is currently established. A value of TRUE indicates the conversation is established; FALSE indicates it is not.

dwTime
Specifies the Windows time at which the conversation was established or terminated. Windows time is the number of milliseconds that have elapsed since the system was started.

hTask
Identifies a task (application instance) that is a partner in the conversation.

hszSvc
Identifies the service name on which the conversation is established.

hszTopic
Identifies the topic name on which the conversation is established.

hConvClient
Identifies the client conversation.

hConvServer
Identifies the server conversation.

See Also

MONCBSTRUCT, MONERRSTRUCT, MONHSZSTRUCT, MONLINKSTRUCT, MONMSGSTRUCT

MONERRSTRUCT

3.1

```
#include <ddeml.h>

typedef struct tagMONERRSTRUCT {  /* mest */
    UINT    cb;
    UINT    wLastError;
    DWORD   dwTime;
    HANDLE  hTask;
} MONERRSTRUCT;
```

The **MONERRSTRUCT** structure contains information about the current dynamic data exchange (DDE) error. A DDE monitoring application can use this structure to monitor errors returned by DDE Management Library functions.

Members

cb
Specifies the length, in bytes, of the structure.

wLastError
Specifies the current error.

dwTime
Specifies the Windows time at which the error occurred. Windows time is the number of milliseconds that have elapsed since the system was started.

hTask
Identifies the task (application instance) that called the DDE function that caused the error.

See Also

MONCBSTRUCT, MONCONVSTRUCT, MONHSZSTRUCT, MONLINKSTRUCT, MONMSGSTRUCT

MONHSZSTRUCT

3.1

```
#include <ddeml.h>

typedef struct tagMONHSZSTRUCT { /* mhst */
    UINT   cb;
    BOOL   fsAction;
    DWORD  dwTime;
    HSZ    hsz;
    HANDLE hTask;
    WORD   wReserved;
    char   str[1];
} MONHSZSTRUCT;
```

The **MONHSZSTRUCT** structure contains information about a dynamic data exchange (DDE) string handle. A DDE monitoring application can use this structure when monitoring the activity of the string-manager component of the DDE Management Library (DDEML).

Members

cb
Specifies the length, in bytes, of the structure.

fsAction
Specifies the action being performed on the string handle identified by the **hsz** member.

Value	Meaning
MH_CLEANUP	An application is freeing its DDE resources, causing the system to delete string handles that the application had created. (The application called the **DdeUninitialize** function.)
MH_CREATE	An application is creating a string handle. (The application called the **DdeCreateStringHandle** function.)
MH_DELETE	An application is deleting a string handle. (The application called the **DdeFreeStringHandle** function.)
MH_KEEP	An application is increasing the use count of a string handle. (The application called the **DdeKeepStringHandle** function.)

dwTime
Specifies the Windows time at which the action specified by the **fsAction** member takes place. Windows time is the number of milliseconds that have elapsed since the system was booted.

hsz
Identifies the string.

hTask
Identifies the task (application instance) performing the action on the string handle.

wReserved
Reserved.

str
Points to the string identified by the **hsz** member.

See Also **MONCBSTRUCT, MONCONVSTRUCT, MONERRSTRUCT, MONLINKSTRUCT, MONMSGSTRUCT**

MONLINKSTRUCT

3.1

```
#include <ddeml.h>

typedef struct tagMONLINKSTRUCT {  /* mlst */
    UINT    cb;
    DWORD   dwTime;
    HANDLE  hTask;
    BOOL    fEstablished;
    BOOL    fNoData;
    HSZ     hszSvc;
    HSZ     hszTopic;
    HSZ     hszItem;
    UINT    wFmt;
    BOOL    fServer;
    HCONV   hConvServer;
    HCONV   hConvClient;
} MONLINKSTRUCT;
```

The **MONLINKSTRUCT** structure contains information about a dynamic data exchange (DDE) advise loop. A DDE monitoring application can use this structure to obtain information about an advise loop that has started or ended.

Members

cb
Specifies the length, in bytes, of the structure.

dwTime
Specifies the Windows time at which the advise loop was started or ended. Windows time is the number of milliseconds that have elapsed since the system was started.

hTask
Identifies a task (application instance) that is a partner in the advise loop.

fEstablished
Indicates whether an advise loop was successfully established. A value of TRUE indicates an advise loop was established; FALSE indicates an advise loop was not established.

fNoData
Indicates whether the XTYPF_NODATA flag was set for the advise loop. A value of TRUE indicates the flag is set; FALSE indicates the flag was not set.

hszSvc
Identifies the service name of the server in the advise loop.

hszTopic
Identifies the topic name on which the advise loop is established.

hszItem
Identifies the item name that is the subject of the advise loop.

wFmt
Specifies the format of the data exchanged (if any) during the advise loop.

fServer
Indicates whether the link notification came from the server. If the notification came from the server, this value is TRUE. Otherwise, it is FALSE.

hConvServer
Identifies the server conversation.

hConvClient
Identifies the client conversation.

See Also **MONCBSTRUCT, MONERRSTRUCT, MONHSZSTRUCT, MONMSGSTRUCT**

MONMSGSTRUCT

3.1

```
#include <ddeml.h>

typedef struct tagMONMSGSTRUCT { /* mmst */
    UINT    cb;
    HWND    hwndTo;
    DWORD   dwTime;
    HANDLE  hTask;
    UINT    wMsg;
    WPARAM  wParam;
    LPARAM  lParam;
} MONMSGSTRUCT;
```

The **MONMSGSTRUCT** structure contains information about a dynamic data exchange (DDE) message. A DDE monitoring application can use this structure to obtain information about a DDE message that was sent or posted.

Members

cb
Specifies the length, in bytes, of the structure.

hwndTo
Identifies the window that receives the DDE message.

dwTime
Specifies the Windows time at which the message was sent or posted. Windows time is the number of milliseconds that have elapsed since the system was started.

hTask
Identifies the task (application instance) containing the window that receives the DDE message.

wMsg
Specifies the identifier of the DDE message.

wParam
Specifies the *wParam* parameter of the DDE message.

lParam
Specifies the *lParam* parameter of the DDE message.

See Also

MONCBSTRUCT, MONCONVSTRUCT, MONERRSTRUCT, MONHSZSTRUCT, MONLINKSTRUCT

MOUSEHOOKSTRUCT

3.1

```
typedef struct tagMOUSEHOOKSTRUCT { /* ms */
    POINT   pt;
    HWND    hwnd;
    UINT    wHitTestCode;
    DWORD   dwExtraInfo;
} MOUSEHOOKSTRUCT;
```

The **MOUSEHOOKSTRUCT** structure contains information about a mouse event.

Members

pt
Specifies a **POINT** structure that contains the x- and y-coordinates of the mouse cursor, in screen coordinates.

hwnd
Identifies the window that will receive the mouse message that corresponds to the mouse event.

wHitTestCode
Specifies the hit-test code.

dwExtraInfo
Specifies extra information associated with the mouse event. An application can set this information by calling the **hardware_event** function and retrieve this information by calling the **GetMessageExtraInfo** function.

See Also **GetMessageExtraInfo**, **SetWindowsHook**

MSG

2.x

```
typedef struct tagMSG {     /* msg */
    HWND   hwnd;
    UINT   message;
    WPARAM wParam;
    LPARAM lParam;
    DWORD  time;
    POINT  pt;
} MSG;
```

The **MSG** structure contains information from the Windows application queue.

Members

hwnd
Identifies the window that receives the message.

message
Specifies the message number.

wParam
Specifies additional information about the message. The exact meaning depends on the **message** value.

lParam
Specifies additional information about the message. The exact meaning depends on the **message** value.

time
Specifies the time at which the message was posted.

pt
Specifies the position of the cursor, in screen coordinates, when the message was posted.

See Also **EVENTMSG**, **GetMessage**

MULTIKEYHELP

3.0

```
typedef struct tagMULTIKEYHELP {    /* mkh */
    UINT  mkSize;
    BYTE  mkKeylist;
    BYTE  szKeyphrase[1];
} MULTIKEYHELP;
```

The **MULTIKEYHELP** structure specifies a keyword table and an associated keyword to be used by the Windows Help application.

Members

mkSize
Specifies the length, in bytes, of the **MULTIKEYHELP** structure.

mkKeylist
Contains a single character that identifies the keyword table to be searched.

szKeyphrase
Contains a null-terminated text string that specifies the keyword to be located in the keyword table.

See Also

WinHelp

NCCALCSIZE_PARAMS

3.1

```
typedef struct tagNCCALCSIZE_PARAMS {
    RECT           rgrc[3];
    WINDOWPOS FAR* lppos;
} NCCALCSIZE_PARAMS;
```

The **NCCALCSIZE_PARAMS** structure contains information that an application can use while processing the WM_NCCALCSIZE message to calculate the size, position, and valid contents of the client area of a window.

Members

rgrc
Specifies an array of rectangles. The first contains the new coordinates of a window that has been moved or resized. The second contains the coordinates of the window before it was moved or resized. The third contains the coordinates of the client area of a window before it was moved or resized. If the window is a child window, the coordinates are relative to the client area of the parent window. If the window is a top-level window, the coordinates are relative to the screen.

lppos

Points to a **WINDOWPOS** structure that contains the size and position values specified in the operation that caused the window to be moved or resized. The **WINDOWPOS** structure has the following form:

```
typedef struct tagWINDOWPOS { /* wp */
    HWND    hwnd;
    HWND    hwndInsertAfter;
    int     x;
    int     y;
    int     cx;
    int     cy;
    UINT    flags;
} WINDOWPOS;
```

See Also **MoveWindow**, **SetWindowPos**, **RECT**, **WINDOWPOS**, WM_NCCALCSIZE

NEWCPLINFO

3.1

```
#include <cpl.h>

typedef struct tagNEWCPLINFO {  /* ncpli */
    DWORD   dwSize;
    DWORD   dwFlags;
    DWORD   dwHelpContext;
    LONG    lData;
    HICON   hIcon;
    char    szName[32];
    char    szInfo[64];
    char    szHelpFile[128];
} NEWCPLINFO;
```

The **NEWCPLINFO** structure contains resource information and a user-defined value for a Control Panel application.

Members

dwSize
Specifies the length of the structure, in bytes.

dwFlags
Specifies Control Panel flags.

dwHelpContext
Specifies the context number for the topic in the help project (.HPJ) file that displays when the user selects help for the application. For more information on help, see *Microsoft Windows Programming Tools*.

lData
Specifies data defined by the application.

hIcon
Identifies an icon resource for the application icon. This icon is displayed in the Control Panel window.

szName
Specifies a null-terminated string that contains the application name. The name is the short string displayed below the application icon in the Control Panel window. The name is also displayed in the Settings menu of Control Panel.

szInfo
Specifies a null-terminated string containing the application description. The description displayed at the bottom of the Control Panel window when the application icon is selected.

szHelpFile
Specifies a null-terminated string that contains the path of the help file, if any, for the application.

NEWTEXTMETRIC

2.x

```
typedef struct tagNEWTEXTMETRIC {    /* ntm */
    int   tmHeight;
    int   tmAscent;
    int   tmDescent;
    int   tmInternalLeading;
    int   tmExternalLeading;
    int   tmAveCharWidth;
    int   tmMaxCharWidth;
    int   tmWeight;
    BYTE  tmItalic;
    BYTE  tmUnderlined;
    BYTE  tmStruckOut;
    BYTE  tmFirstChar;
    BYTE  tmLastChar;
    BYTE  tmDefaultChar;
    BYTE  tmBreakChar;
    BYTE  tmPitchAndFamily;
    BYTE  tmCharSet;
    int   tmOverhang;
    int   tmDigitizedAspectX;
    int   tmDigitizedAspectY;
    DWORD ntmFlags;
    UINT  ntmSizeEM;
    UINT  ntmCellHeight;
    UINT  ntmAvgWidth;
} NEWTEXTMETRIC;
```

The **NEWTEXTMETRIC** structure contains basic information about a physical font. The last four members of the **NEWTEXTMETRIC** structure are not included in the **TEXTMETRIC** structure; in all other respects, the structures are identical. The additional members are used for information about TrueType fonts.

Members

tmHeight
Specifies the height of character cells. (The height is the sum of the **tmAscent** and **tmDescent** members.)

tmAscent
Specifies the ascent of character cells. (The ascent is the space between the base line and the top of the character cell.)

tmDescent
Specifies the descent of character cells. (The descent is the space between the bottom of the character cell and the base line.)

tmInternalLeading
Specifies the difference between the point size of a font and the physical size of the font. For TrueType fonts, this value is equal to **tmHeight** minus (*s* * **ntmSizeEM**), where *s* is the scaling factor for the TrueType font. For

bitmap fonts, this value is used to determine the point size of a font; when an application specifies a negative value in the **lfHeight** member of the **LOGFONT** structure, the application is requesting a font whose height equals **tmHeight** minus **tmInternalLeading**.

tmExternalLeading
Specifies the amount of extra leading (space) that the application adds between rows. Since this area is outside the character cell, it contains no marks and will not be altered by text output calls in either opaque or transparent mode. The font designer sometimes sets this member to zero.

tmAveCharWidth
Specifies the average width of characters in the font. For ANSI_CHARSET fonts, this is a weighted average of the characters "a" through "z" and the space character. For other character sets, this value is an unweighted average of all characters in the font.

tmMaxCharWidth
Specifies the "B" spacing of the widest character in the font. For more information about "B" spacing, see the description of the **ABC** structure.

tmWeight
Specifies the weight of the font. This member can be one of the following values:

Constant	Value
FW_DONTCARE	0
FW_THIN	100
FW_EXTRALIGHT	200
FW_ULTRALIGHT	200
FW_LIGHT	300
FW_NORMAL	400
FW_REGULAR	400
FW_MEDIUM	500
FW_SEMIBOLD	600
FW_DEMIBOLD	600
FW_BOLD	700
FW_EXTRABOLD	800
FW_ULTRABOLD	800
FW_BLACK	900
FW_HEAVY	900

tmItalic
Specifies an italic font if it is nonzero.

tmUnderlined
Specifies an underlined font if it is nonzero.

tmStruckOut
Specifies a "struckout" font if it is nonzero.

tmFirstChar
Specifies the value of the first character defined in the font.

tmLastChar
Specifies the value of the last character defined in the font.

tmDefaultChar
Specifies the value of the character that will be substituted for characters 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 four low-order bits identify the type of font, as follows:

Value	Meaning
TMPF_PITCH	Designates a fixed-pitch font.
TMPF_VECTOR	Designates a vector or TrueType font.
TMPF_TRUETYPE	Designates a TrueType font.
TMPF_DEVICE	Designates a device font.

Some fonts are identified by several of these bits—for example, the bits TMPF_PITCH, TMPF_VECTOR, and TMPF_TRUETYPE would be set for the monospace TrueType font, Courier New. The TMPF_DEVICE bit could be set for a TrueType font as well, because this bit is set for both downloaded and device-resident fonts.

When the TMPF_TRUETYPE bit is set, the font is usable on all output devices. For example, if a TrueType font existed on a printer but could not be used on the display, the TMPF_TRUETYPE bit would not be set for that font.

The four high-order bits specify the font family. The **tmPitchAndFamily** member can be combined with the hexadecimal value 0xF0 by using the bitwise AND operator and can then be compared with the font family names for an identical match. The following font families are defined:

Value	Meaning
FF_DECORATIVE	Novelty fonts. Old English is an example.
FF_DONTCARE	Don't care or don't know.
FF_MODERN	Fonts with constant stroke width, with or without serifs. Pica, Elite, and Courier New are examples.
FF_ROMAN	Fonts with variable stroke width and with serifs. Times New Roman and New Century Schoolbook are examples.

Value	Meaning
FF_SCRIPT	Fonts designed to look like handwriting. Script and Cursive are examples.
FF_SWISS	Fonts with variable stroke width and without serifs. MS Sans Serif is an example.

tmCharSet
Specifies the character set of the font. The following values are defined:

Constant	Value
ANSI_CHARSET	0
DEFAULT_CHARSET	1
SYMBOL_CHARSET	2
SHIFTJIS_CHARSET	128
OEM_CHARSET	255

tmOverhang
Specifies the extra width that is added to some synthesized fonts. When synthesizing some attributes, such as bold or italic, graphics-device interface (GDI) or a device adds width to a string on both a per-character and per-string basis. For example, GDI makes a string bold by expanding the intracharacter spacing and overstriking by an offset value and italicizes a font by skewing the string. In either case, the string is wider after the attribute is synthesized. For bold strings, the overhang is the distance by which the overstrike is offset. For italic strings, the overhang is the amount the top of the font is skewed past the bottom of the font.

The **tmOverhang** member is zero for many italic and bold TrueType fonts because many TrueType fonts include italic and bold faces that are not synthesized. For example, the overhang for Courier New Italic is zero.

An application that uses raster fonts can use the overhang value to determine the spacing between words that have different attributes.

tmDigitizedAspectX
Specifies the horizontal aspect of the device for which the font was designed.

tmDigitizedAspectY
Specifies the vertical aspect of the device for which the font was designed. The ratio of the **tmDigitizedAspectX** and **tmDigitizedAspectY** members is the aspect ratio of the device for which the font was designed.

ntmFlags
Specifies some elements of the font style. This member can be one or more of the following values:

NTM_REGULAR
NTM_BOLD
NTM_ITALIC

The NTM_BOLD and NTM_ITALIC flags could be combined with the OR operator to specify a bold italic font.

ntmSizeEM
Specifies the size of the em square for the font, in the units for which the font was designed (notional units).

ntmCellHeight
Specifies the height of the font, in the units for which the font was designed (notional units). This value should be compared against the value of the **ntmSizeEM** member.

ntmAvgWidth
Specifies the average width of characters in the font, in the units for which the font was designed (notional units). This value should be compared against the value of the **ntmSizeEM** member.

Comments

The sizes in the **NEWTEXTMETRIC** structure are typically given in logical units; that is, they depend on the current mapping mode of the display context.

See Also

EnumFontFamilies, EnumFonts, GetDeviceCaps, GetTextMetrics

NFYLOADSEG

3.1

```
#include <toolhelp.h>

typedef struct tagNFYLOADSEG {  /* nfyls */
    DWORD   dwSize;
    WORD    wSelector;
    WORD    wSegNum;
    WORD    wType;
    WORD    wcInstance;
    LPCSTR  lpstrModuleName;
} NFYLOADSEG;
```

The **NFYLOADSEG** structure contains information about the segment being loaded when the kernel sends a load-segment notification.

Members

dwSize
Specifies the size of the **NFYLOADSEG** structure, in bytes.

wSelector
Contains the selector of the segment being loaded.

wSegNum
Contains the executable-file segment number.

wType
Indicates the type of information in the segment. Only the low bit of **wType** is used. This type can be one of the following values:

Value	Meaning
0	The segment contains code.
1	The segment contains data.

wcInstance
Specifies the number of instances that share this segment. This value is valid only for data segments.

lpstrModuleName
Points to a null-terminated string containing the name of the module that owns the segment being loaded.

See Also **NotifyRegister**

NFYLOGERROR

3.1

```
#include <toolhelp.h>

typedef struct tagNFYLOGERROR {  /* nfyle */
    DWORD     dwSize;
    UINT      wErrCode;
    void FAR* lpInfo;
} NFYLOGERROR;
```

The **NFYLOGERROR** structure contains information about a validation error that caused the kernel to send an NFY_LOGERROR notification.

Members

dwSize
Specifies the size of the **NFYLOGERROR** structure, in bytes.

wErrCode
Identifies the error value that caused the notification to be sent.

lpInfo
Points to additional information, dependent on the error value.

See Also **NotifyRegister**

NFYLOGPARAMERROR

3.1

```
#include <toolhelp.h>

typedef struct tagNFYLOGPARAMERROR {  /* nfylpe */
    DWORD          dwSize;
    UINT           wErrCode;
    FARPROC        lpfnErrorAddr;
    void FAR* FAR* lpBadParam;
} NFYLOGPARAMERROR;
```

The **NFYLOGPARAMERROR** structure contains information about a parameter-validation error that caused the kernel to send an NFY_LOGPARAMERROR notification.

Members

dwSize
Specifies the size of the **NFYLOGPARAMERROR** structure, in bytes.

wErrCode
Identifies the error value that caused the notification to be sent.

lpfnErrorAddr
Points to the address of the function with the invalid parameter.

lpBadParam
Points to the name of the invalid parameter.

See Also

NotifyRegister

NFYRIP

3.1

```
#include <toolhelp.h>

typedef struct tagNFYRIP {  /* nfyr */
    DWORD dwSize;
    WORD wIP;
    WORD wCS;
    WORD wSS;
    WORD wBP;
    WORD wExitCode;
} NFYRIP;
```

The **NFYRIP** structure contains information about the system when a system debugging error (RIP) occurs.

Members

dwSize
Specifies the size of the **NFYRIP** structure, in bytes.

wIP
Contains the value in the IP register at the time of the RIP.

wCS
Contains the value in the CS register at the time of the RIP.

wSS
Contains the value in the SS register at the time of the RIP.

wBP
Contains the value in the BP register at the time of the RIP.

wExitCode
Contains an exit code that describes why the RIP occurred.

Comments

The **StackTraceCSIPFirst** function uses the CS:IP and SS:BP values presented in this structure. The first frame in the stack identified by these values points to the **FatalExit** function. The next frame points to the routine that called **FatalExit**, usually in USER.EXE, GDI.EXE, or either KRNL286.EXE or KRNL386.EXE.

See Also

FatalExit, **NotifyRegister**, **StackTraceCSIPFirst**

NFYSTARTDLL

3.1

```
#include <toolhelp.h>

typedef struct tagNFYSTARTDLL {  /* nfysd */
    DWORD   dwSize;
    HMODULE hModule;
    WORD    wCS;
    WORD    wIP;
} NFYSTARTDLL;
```

The **NFYSTARTDLL** structure contains information about the dynamic-link library (DLL) being loaded when the kernel sends a load-DLL notification.

Members

dwSize
Specifies the size of the **NFYSTARTDLL** structure, in bytes.

hModule
Identifies the library module being loaded.

wCS
Contains the value in the CS register at load time. This value is used with the value of the **wIP** member to determine the load address of the library.

wIP
Contains the value in the IP register at load time. This value is used with the **wCS** value to determine the load address of the library.

See Also **NotifyRegister**

OFSTRUCT

2.x

```
typedef struct tagOFSTRUCT {    /* of */
   BYTE  cBytes;
   BYTE  fFixedDisk;
   UINT  nErrCode;
   BYTE  reserved[4];
   BYTE  szPathName[128];
} OFSTRUCT;
```

The **OFSTRUCT** structure contains file information which results from opening that file.

Members

cBytes
Specifies the length, in bytes, of the **OFSTRUCT** structure.

fFixedDisk
Specifies whether the file is on a fixed disk. The **fFixedDisk** member is non-zero if the file is on a fixed disk.

nErrCode
Specifies the MS-DOS error value if the **OpenFile** function returns –1 (that is, **OpenFile** fails). For a list of possible error values, see the following Comments section.

reserved
Reserved member. Four bytes reserved for future use.

szPathName
Specifies 128 bytes that contain the path of the file. This string consists of characters from the OEM character set.

Comments

The error values that may be specified in the **nErrCode** parameter follow:

Value	Meaning
0x0001	Invalid function
0x0002	File not found
0x0003	Path not found
0x0004	Too many open files
0x0005	Access denied
0x0006	Invalid handle
0x0007	Arena trashed
0x0008	Not enough memory
0x0009	Invalid block
0x000A	Bad environment
0x000B	Bad format
0x000C	Invalid access
0x000D	Invalid data
0x000F	Invalid drive
0x0010	Current directory
0x0011	Not same device
0x0012	No more files
0x0013	Write protect error
0x0014	Bad unit
0x0015	Not ready
0x0016	Bad command
0x0017	CRC error
0x0018	Bad length
0x0019	Seek error
0x001A	Not MS-DOS disk
0x001B	Sector not found
0x001C	Out of paper
0x001D	Write fault
0x001E	Read fault
0x001F	General failure
0x0020	Sharing violation
0x0021	Lock violation
0x0022	Wrong disk
0x0023	File control block unavailable
0x0024	Sharing buffer exceeded
0x0032	Not supported
0x0033	Remote not listed
0x0034	Duplicate name

Value	Meaning
0x0035	Bad netpath
0x0036	Network busy
0x0037	Device does not exist
0x0038	Too many commands
0x0039	Adaptor hardware error
0x003A	Bad network response
0x003B	Unexpected network error
0x003C	Bad remote adaptor
0x003D	Print queue full
0x003E	No spool space
0x003F	Print canceled
0x0040	Netname deleted
0x0041	Network access denied
0x0042	Bad device type
0x0043	Bad network name
0x0044	Too many names
0x0045	Too many sessions
0x0046	Sharing paused
0x0047	Request not accepted
0x0048	Redirection paused
0x0050	File exists
0x0051	Duplicate file control block
0x0052	Cannot make
0x0053	Interrupt 24 failure
0x0054	Out of structures
0x0055	Already assigned
0x0056	Invalid password
0x0057	Invalid parameter
0x0058	Net write fault

See Also

OpenFile

OLECLIENT

3.1

```
#include <ole.h>

typedef struct _OLECLIENT { /* oc */
    LPOLECLIENTVTBL lpvtbl;
    .
    .  /* any client-supplied state information */
    .
} OLECLIENT;
```

The **OLECLIENT** structure points to an **OLECLIENTVTBL** structure and can store state information for use by the client application.

Members

lpvtbl
Points to a table of function pointers for the client.

Comments

Servers and object handlers should not attempt to use any state information supplied in the **OLECLIENT** structure. The use and meaning of this information is entirely dependent on the client application. Because a pointer to this structure is supplied as a parameter to the client's callback function, this is the preferred method for the client application to store private object-state information.

OLECLIENTVTBL

3.1

```
#include <ole.h>

typedef struct _OLECLIENTVTBL {     /* ocv */
    int (CALLBACK* CallBack)(LPOLECLIENT, OLE_NOTIFICATION,
LPOLEOBJECT);
} OLECLIENTVTBL;
```

The **OLECLIENTVTBL** structure contains a pointer to a callback function for the client application.

Comments

The address passed as the **CallBack** member must be created by using the **MakeProcInstance** function.

Function ClientCallback

INT ClientCallback(*lpclient*, *notification*, *lpobject*)
LPOLECLIENT *lpclient*;
OLE_NOTIFICATION *notification*;
LPOLEOBJECT *lpobject*;

The **ClientCallback** function must use the Pascal calling convention and must be declared **FAR**.

Parameters

lpclient

Points to the client structure associated with the object. The library retrieves this pointer from its object structure when a notification occurs, uses it to locate the callback function, and passes the pointer to the client structure for the client application's use.

notification

Specifies the reason for the notification. This parameter can be one of the following values:

Value	Meaning
OLE_CHANGED	The linked object has changed. (This notification is not sent for embedded objects.) A typical action to take with this notification is either to redraw or to save the object.
OLE_CLOSED	The object has been closed in its server. When the client receives this notification, it should not call any function that causes an asynchronous operation until it regains control of program execution.
OLE_QUERY_PAINT	A lengthy drawing operation is occurring. This notification allows the drawing to be interrupted.
OLE_QUERY_RETRY	The server has responded to a request by indicating that it is busy. This notification requests the client to determine whether the library should continue to make the request. If the callback function returns FALSE, the transaction with the server is discontinued.
OLE_RELEASE	The object has been released because an asynchronous operation has finished. The client should not quit until all objects have been released. The client application can call the **OleQueryReleaseError** function to determine whether the operation succeeded. It can also call the **OleQueryReleaseMethod** function, if necessary, to verify that that operation has ended

Value	Meaning
OLE_RENAMED	The linked object has been renamed in its server. This notification is for information only, because the library automatically updates its link information.
OLE_SAVED	The linked object has been saved in its server. The client receives this notification when the server calls the **OleSavedServerDoc** function in response to the user choosing the Update command in the server's File menu.

When the client receives the **OLE_CLOSED** notification, it typically stores the condition and returns to the client library, taking action only when the client library returns control of program execution to the client application. If the client application must take action before regaining control, it should not call any functions that could result in an asynchronous operation.

lpobject
Points to the object that caused the notification to be sent. Applications that use the same client structure for more than one object use the *lpobject* parameter to distinguish between notifications.

Return Value

When the *notification* parameter specifies either OLE_QUERY_PAINT or OLE_QUERY_RETRY, the client should return TRUE if the library should continue, or FALSE to terminate the painting operation or discontinue the server transaction. When the *notification* parameter does not specify either OLE_QUERY_PAINT or OLE_QUERY_RETRY, the return value is ignored.

Comments

The client application should act on these notifications at the next appropriate time; for example, as part of the main event loop or when closing the object. The updating of an object can be deferred until the user requests the update, if the client provides that functionality. The client may call the library from a notification callback function (the library is reentrant). The client should not attempt an asynchronous operation while certain other operations are in progress (for example, opening or deleting an object). The client also should not enter a message-dispatch loop inside the callback function. When the client application calls a function that would cause an asynchronous operation, the client library returns OLE_WAIT_FOR_RELEASE when the function is called, notifies the application when the operation completes by using OLE_RELEASE, and returns OLE_BUSY if the client attempts to invoke a conflicting operation while the previous one is in progress. The client can determine if an asynchronous operation is in progress by calling **OleQueryReleaseStatus**, which returns OLE_BUSY if the operation has not yet completed.

See Also

OleQueryReleaseStatus

OLEOBJECT

3.1

```
#include <ole.h>

typedef struct _OLEOBJECT {      /* oo */
    LPOLEOBJECTVTBL lpvtbl;
    .
    .  /* any server-supplied state information */
    .
} OLEOBJECT;
```

The **OLEOBJECT** structure points to a table of function pointers for an object. This structure is initialized and maintained by servers for the server library.

Members

lpvtbl
Points to a table of function pointers for the object.

OLEOBJECTVTBL

3.1

```
#include <ole.h>

typedef struct _OLEOBJECTVTBL {    /* oov */
    void FAR* (CALLBACK* QueryProtocol)(LPOLEOBJECT, OLE_LPCSTR);
    OLESTATUS (CALLBACK* Release)(LPOLEOBJECT);
    OLESTATUS (CALLBACK* Show)(LPOLEOBJECT, BOOL);
    OLESTATUS (CALLBACK* DoVerb)(LPOLEOBJECT, UINT, BOOL, BOOL);
    OLESTATUS (CALLBACK* GetData)(LPOLEOBJECT, OLECLIPFORMAT,
        HANDLE FAR*);
    OLESTATUS (CALLBACK* SetData)(LPOLEOBJECT, OLECLIPFORMAT, HANDLE);
    OLESTATUS (CALLBACK* SetTargetDevice)(LPOLEOBJECT, HGLOBAL);
    OLESTATUS (CALLBACK* SetBounds)(LPOLEOBJECT, OLE_CONST RECT FAR*);
    OLECLIPFORMAT (CALLBACK* EnumFormats)(LPOLEOBJECT, OLECLIPFORMAT);
    OLESTATUS (CALLBACK* SetColorScheme)(LPOLEOBJECT,
        OLE_CONST LOGPALETTE FAR*);

    /*
     * Server applications implement only the functions listed above.
     * Object handlers can use any of the functions in this structure
     * to modify default server behavior.
     */
```

```
    OLESTATUS (CALLBACK* Delete)(LPOLEOBJECT);
    OLESTATUS (CALLBACK* SetHostNames)(LPOLEOBJECT, OLE_LPCSTR,
        OLE_LPCSTR);
    OLESTATUS (CALLBACK* SaveToStream)(LPOLEOBJECT, LPOLESTREAM);
    OLESTATUS (CALLBACK* Clone)(LPOLEOBJECT, LPOLECLIENT, LHCLIENTDOC,
        OLE_LPCSTR, LPOLEOBJECT FAR*);
    OLESTATUS (CALLBACK* CopyFromLink)(LPOLEOBJECT, LPOLECLIENT,
        LHCLIENTDOC, OLE_LPCSTR, LPOLEOBJECT FAR*);
    OLESTATUS (CALLBACK* Equal)(LPOLEOBJECT, LPOLEOBJECT);
    OLESTATUS (CALLBACK* CopyToClipboard)(LPOLEOBJECT);
    OLESTATUS (CALLBACK* Draw)(LPOLEOBJECT, HDC, OLE_CONST RECT FAR*,
        OLE_CONST RECT FAR*, HDC);
    OLESTATUS (CALLBACK* Activate)(LPOLEOBJECT, UINT, BOOL, BOOL, HWND,
        OLE_CONST RECT FAR*);
    OLESTATUS (CALLBACK* Execute)(LPOLEOBJECT, HGLOBAL, UINT);
    OLESTATUS (CALLBACK* Close)(LPOLEOBJECT);
    OLESTATUS (CALLBACK* Update)(LPOLEOBJECT);
    OLESTATUS (CALLBACK* Reconnect)(LPOLEOBJECT);
    OLESTATUS (CALLBACK* ObjectConvert)(LPOLEOBJECT, OLE_LPCSTR,
        LPOLECLIENT, LHCLIENTDOC, OLE_LPCSTR, LPOLEOBJECT FAR*);
    OLESTATUS (CALLBACK* GetLinkUpdateOptions)(LPOLEOBJECT,
        OLEOPT_UPDATE FAR*);
    OLESTATUS (CALLBACK* SetLinkUpdateOptions)(LPOLEOBJECT,
        OLEOPT_UPDATE);
    OLESTATUS (CALLBACK* Rename)(LPOLEOBJECT, OLE_LPCSTR);
    OLESTATUS (CALLBACK* QueryName)(LPOLEOBJECT, LPSTR, UINT FAR*);
    OLESTATUS (CALLBACK* QueryType)(LPOLEOBJECT, LONG FAR*);
    OLESTATUS (CALLBACK* QueryBounds)(LPOLEOBJECT, RECT FAR*);
    OLESTATUS (CALLBACK* QuerySize)(LPOLEOBJECT, DWORD FAR*);
    OLESTATUS (CALLBACK* QueryOpen)(LPOLEOBJECT);
    OLESTATUS (CALLBACK* QueryOutOfDate)(LPOLEOBJECT);
    OLESTATUS (CALLBACK* QueryReleaseStatus)(LPOLEOBJECT);
    OLESTATUS (CALLBACK* QueryReleaseError)(LPOLEOBJECT);
    OLE_RELEASE_METHOD (CALLBACK* QueryReleaseMethod)(LPOLEOBJECT);
    OLESTATUS (CALLBACK* RequestData)(LPOLEOBJECT, OLECLIPFORMAT);
    OLESTATUS (CALLBACK* ObjectLong)(LPOLEOBJECT, UINT, LONG FAR*);
} OLEOBJECTVTBL;
```

The **OLEOBJECTVTBL** structure points to functions that manipulate an object. A server application creates this structure and an **OLEOBJECT** structure to give the server library access to an object.

Server applications do not need to implement functions beyond the **SetColorScheme** function. Object handlers can provide specialized treatment for some or all of the functions in the **OLEOBJECTVTBL** structure.

The following list of structure members does not document all the functions pointed to by the **OLEOBJECTVTBL** structure. For information about the functions not documented here, see the documentation for the corresponding function for object linking and embedding (OLE). For example, for more information about the **QueryProtocol** member, see the **OleQueryProtocol** function.

Comments

The following functions in **OLEOBJECTVTBL** should return OLE_BUSY when appropriate:

Activate	**SetBounds**
Close	**SetColorScheme**
CopyFromLink	**SetData**
Delete	**SetHostNames**
DoVerb	**SetLinkUpdateOptions**
Execute	**SetTargetDevice**
ObjectConvert	**Show**
Reconnect	**Update**
RequestData	

Function

Release

OLESTATUS (FAR PASCAL *Release)(*lpObject*)
LPOLEOBJECT *lpObject*;

The **Release** function causes the server to free the resources associated with the specified **OLEOBJECT** structure.

Parameters

lpObject
Points to the **OLEOBJECT** structure to be released.

Return Value

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

Comments

The server application should not destroy data when the library calls the **Release** function. The library calls the **Release** function when no clients are connected to the object.

Function

Show

OLESTATUS (FAR PASCAL *Show)(*lpObject*, *fTakeFocus*)
LPOLEOBJECT *lpObject*;
BOOL *fTakeFocus*;

The **Show** function causes the server to show an object, displaying its window and scrolling (if necessary) to make the object visible.

Parameters

lpObject
Points to the **OLEOBJECT** structure to show.

fTakeFocus
Specifies whether the server window gets the focus. If the server window is to get the focus, this value is TRUE. Otherwise, this value is FALSE.

Return Value

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

Comments

The library calls the **Show** function when the server application should show the document to the user for editing or to request the server to scroll the document to bring the object into view.

Function **DoVerb**

OLESTATUS (FAR PASCAL *DoVerb)(*lpObject*, *iVerb*, *fShow*, *fTakeFocus*);
LPOLEOBJECT *lpObject*;
UINT *iVerb*;
BOOL *fShow*;
BOOL *fTakeFocus*;

The **DoVerb** function specifies what kind of action the server should take when a user activates an object.

Parameters

lpObject
Points to the object to activate.

iVerb
Specifies the action to take. The meaning of this parameter is determined by the server application.

fShow
Specifies whether to show the server window. This value is TRUE to show the window; otherwise, it is FALSE.

fTakeFocus
Specifies whether the server window gets the focus. If the server window is to get the focus, this value is TRUE. Otherwise, it is FALSE. This parameter is relevant only if the *fShow* parameter is TRUE.

Return Value

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

Comments

All servers must support the editing of objects. If a server does not support any verbs except Edit, it should edit the object no matter what value is specified by the *iVerb* parameter.

Function GetData

OLESTATUS (FAR PASCAL *GetData)(*lpObject*, *cfFormat*, *lphdata*)
LPOLEOBJECT *lpObject*;
OLECLIPFORMAT *cfFormat*;
HANDLE FAR* *lphdata*;

The **GetData** function retrieves data from an object in a specified format. The server application should allocate memory, fill it with the data, and return the data through the *lphdata* parameter.

Parameters

lpObject
Points to the **OLEOBJECT** structure from which data is requested.

cfFormat
Specifies the format in which the data is requested.

lphdata
Points to the handle of the allocated memory that the server application returns. The library frees the memory when it is no longer needed.

Return Value

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

OLE_ERROR_BLANK
OLE_ERROR_FORMAT
OLE_ERROR_OBJECT

Function SetData

OLESTATUS (FAR PASCAL *SetData)(*lpObject*, *cfFormat*, *hdata*)
LPOLEOBJECT *lpObject*;
OLECLIPFORMAT *cfFormat*;
HANDLE *hdata*;

The **SetData** function stores data in an object in a specified format. This function is called (with the Native data format) when a client opens an embedded object for editing. This function is also used if the client calls the **OleSetData** function with some other format.

Parameters

lpObject
Points to the **OLEOBJECT** structure in which data is stored.

cfFormat
Specifies the format of the data.

hdata
Identifies a place in memory from which the server application should extract the data. The server should delete this handle after it uses the data.

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

Comments The server application is responsible for the memory identified by the *hdata* parameter. The server must delete this data even if it returns OLE_BUSY or if an error occurs.

Function SetTargetDevice

OLESTATUS (FAR PASCAL *SetTargetDevice)(*lpObject*, *hotd*)
LPOLEOBJECT *lpObject*;
HGLOBAL *hotd*;

The **SetTargetDevice** function communicates information about the client's target device for the object. The server can use this information to customize output for the target device.

Parameters

lpObject
Points to the **OLEOBJECT** structure for which the target device is specified.

hotd
Identifies an **OLETARGETDEVICE** structure.

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

Comments The server application is responsible for the memory identified by the *hotd* parameter. The server must delete this data even if it returns OLE_BUSY or if an error occurs.

The library passes NULL for the *hotd* parameter to indicate that the rendering is necessary for the screen.

See Also **OleSetTargetDevice**

Function ObjectLong

OLESTATUS (FAR PASCAL *ObjectLong)(*lpObject*, *wFlags*, *lpData*)
LPOLEOBJECT *lpObject*;
UINT *wFlags*;
LONG FAR* *lpData*;

The **ObjectLong** function allows the calling application to store data with an object. This function is typically used by object handlers.

Parameters

lpObject
Points to the **OLEOBJECT** structure for which the data is stored.

wFlags
Specifies the method used for setting and retrieving data. It can be one or more of the following values:

Value	Meaning
OF_SET	Data is written to the location specified by the *lpData* parameter, replacing any data already there.
OF_GET	Data is read from the location specified by the *lpData* parameter.
OF_HANDLER	Data is written or read by an object handler. This value prevents data from an object handler from being replaced by other applications.

If the calling application specifies OF_SET and OF_GET, the function returns a pointer to the previous data and replaces the data pointed to by the *lpData* parameter with the data specified by the calling application.

lpData
Points to data to be written or read.

Return Value

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

Function

SetColorScheme

OLESTATUS SetColorScheme(*lpObject***, *lpPal*)**
LPOLEOBJECT *lpObject*;
OLE_CONST LOGPALETTE FAR* *lpPal*;

The **SetColorScheme** function sends the server application the color palette recommended by the client application.

Parameters

lpObject
Points to an **OLEOBJECT** structure for which the client application recommends a palette.

lpPal
Points to a **LOGPALETTE** structure specifying the recommended palette.

Return Value

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

Comments

Server applications are not required to use the palette recommended by the client application.

Before returning from the **SetColorScheme** function, the server application should use the palette pointed to by the *lpPal* parameter in a call to the CreatePalette function to create the handle of the palette:

```
hpal = CreatePalette(lpPal);
```

The server can then use the palette handle to refer to the palette.

The first palette entry in the **LOGPALETTE** structure specifies the foreground color recommended by the client application. The second palette entry specifies the background color. The first half of the remaining palette entries are fill colors, and the second half are colors for lines and text.

Client applications typically specify an even number of palette entries. When there is an uneven number of entries, the server should interpret the odd entry as a fill color; that is, if there are five entries, three should be interpreted as fill colors and two as line and text colors.

OLESERVER

3.1

```
#include <ole.h>

typedef struct _OLESERVER {     /* os */
    LPOLESERVERVTBL lpvtbl;
    .
    .  /* any server-supplied state information */
    .
} OLESERVER;
```

The **OLESERVER** structure points to a table of function pointers for the server. This structure is initialized and maintained by servers for the server library.

Members

lpvtbl
Points to a table of function pointers for the server.

OLESERVERDOC

3.1

```
#include <ole.h>

typedef struct _OLESERVERDOC {  /* osd */
    LPOLESERVERDOCVTBL lpvtbl;
    .
    .  /* any server-supplied document-state information */
    .

} OLESERVERDOC;
```

The **OLESERVERDOC** structure points to a table of function pointers for a document. This structure is initialized and maintained by servers for the server library.

Members

lpvtbl
Points to a table of function pointers for the document.

OLESERVERDOCVTBL

3.1

```
#include <ole.h>

typedef struct _OLESERVERDOCVTBL {   /* odv */
    OLESTATUS (CALLBACK* Save)(LPOLESERVERDOC);
    OLESTATUS (CALLBACK* Close)(LPOLESERVERDOC);
    OLESTATUS (CALLBACK* SetHostNames)(LPOLESERVERDOC, OLE_LPCSTR,
        OLE_LPCSTR);
    OLESTATUS (CALLBACK* SetDocDimensions)(LPOLESERVERDOC,
        OLE_CONST RECT FAR*);
    OLESTATUS (CALLBACK* GetObject)(LPOLESERVERDOC, OLE_LPCSTR,
        LPOLEOBJECT FAR*, LPOLECLIENT);
    OLESTATUS (CALLBACK* Release)(LPOLESERVERDOC);
    OLESTATUS (CALLBACK* SetColorScheme)(LPOLESERVERDOC,
        OLE_CONST LOGPALETTE FAR*);
    OLESTATUS (CALLBACK* Execute)(LPOLESERVERDOC, HGLOBAL);
} OLESERVERDOCVTBL;
```

The **OLESERVERDOCVTBL** structure points to functions that manipulate a document. A server application creates this structure and an **OLESERVERDOC** structure to give the server library access to a document.

Documents opened or created on request from the library should not be shown to the user for editing until the library requests that they be shown.

Every function except **Release** can return OLE_BUSY.

Function Save

OLESTATUS Save(*lpDoc***)**
LPOLESERVERDOC *lpDoc***;**

The **Save** function instructs the server to save the document.

Parameters

lpDoc
Points to an **OLESERVERDOC** structure corresponding to the document to save.

Return Value

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

Function Close

OLESTATUS Close(*lpDoc***)**
LPOLESERVERDOC *lpDoc***;**

The **Close** function instructs the server application to unconditionally close the document. The library calls this function when the client application initiates the closure.

Parameters

lpDoc
Points to an **OLESERVERDOC** structure corresponding to the document to close.

Return Value

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

Comments

The library always calls the **Close** function before calling the **Release** function in the **OLESERVERVTBL** structure.

The server application should not prompt the user to save the document or take other actions; messages of this kind are handled by the client application.

When the library calls the **Close** function, the server should respond by calling the **OleRevokeServerDoc** function. The resources for the document are freed when the library calls the **Release** function. The server should not wait for the **Release** function by entering a message-dispatch loop after calling **OleRevokeServerDoc**. (A server should never enter message-dispatch loops while processing any of these functions.)

When a document is closed, the server should free the memory for the **OLESERVERDOCVTBL** structure and associated resources.

Function SetHostNames

OLESTATUS SetHostNames(*lpDoc*, *lpszClient*, *lpszDoc*)
LPOLESERVERDOC *lpDoc*;
OLE_LPCSTR *lpszClient*;
OLE_LPCSTR *lpszDoc*;

The **SetHostNames** function sets the name that should be used for a window title. This name is used only for an embedded object, because a linked object has its own title. This function is used only for documents that are embedded objects.

Parameters

lpDoc
Points to an **OLESERVERDOC** structure corresponding to a document that is the embedded object for which a name is specified.

lpszClient
Points to a null-terminated string specifying the name of the client.

lpszDoc
Points to a null-terminated string specifying the client's name for the object.

Return Value

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

Function SetDocDimensions

OLESTATUS SetDocDimensions(*lpDoc*, *lpRect*)
LPOLESERVERDOC *lpDoc*;
OLE_CONST RECT FAR* *lpRect*;

The **SetDocDimensions** function gives the server the rectangle on the target device for which the object should be formatted. This function is relevant only for documents that are embedded objects.

Parameters

lpDoc
Points to the **OLESERVERDOC** structure corresponding to the document that is the embedded object for which the target size is specified.

lpRect
Points to a **RECT** structure containing the target size of the object, in MM_HIMETRIC units. (In the MM_HIMETRIC mapping mode, the positive y-direction is up.)

Return Value

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

Function GetObject

OLESTATUS GetObject(*lpDoc*, *lpszItem*, *lplpObject*, *lpClient*)
LPOLESERVERDOC *lpDoc*;
OLE_LPCSTR *lpszItem*;
LPOLEOBJECT FAR* *lplpObject*;
LPOLECLIENT *lpClient*;

The **GetObject** function requests the server to create an **OLEOBJECT** structure.

Parameters

lpDoc
Points to an **OLESERVERDOC** structure corresponding to this document.

lpszItem
Points to a null-terminated string specifying the name of an item in the specified document for which an object structure is requested. If this string is set to NULL, the entire document is requested. This string cannot contain a slash mark (/).

lplpObject
Points to a variable of type **LPOLEOBJECT** in which the server application should return a long pointer to the allocated **OLEOBJECT** structure.

lpClient
Points to an **OLECLIENT** structure allocated by the library. The server should associate the **OLECLIENT** structure with the object and use it to notify the library of changes to the object.

Return Value

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

Comments

The server application should allocate and initialize the **OLEOBJECT** structure, associate it with the **OLECLIENT** structure pointed to by the *lpClient* parameter, and return a pointer to the **OLEOBJECT** structure through the *lplpObject* argument.

The library calls the **GetObject** function to associate a client with the part of the document identified by the *lpszItem* parameter. When a client has been associated with an object by this function, the server can send notifications to the client.

Applications should be prepared to handle multiple calls to **GetObject** for a given object. This entails creating multiple **OLECLIENT** structures and sending notifications to each of these structures when appropriate. Multiple calls to **GetObject** are possible because some client applications that implement object linking and embedding (OLE) by using dynamic data exchange (DDE) rather than the OLE dynamic-link libraries may use both NULL and an actual item name for the *lpszItem* parameter.

Function Release

OLESTATUS Release(*lpDoc***)**
LPOLESERVERDOC *lpDoc*;

The **Release** function notifies the server when a revoked document has terminated conversations and can be destroyed.

Parameters

lpDoc
Points to an **OLESERVERDOC** structure for which the handle was revoked and which can now be released.

Return Value

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

Function SetColorScheme

OLESTATUS SetColorScheme(*lpDoc***, *lpPal*)**
LPOLESERVERDOC *lpDoc*;
OLE_CONST LOGPALETTE FAR* *lpPal*;

The **SetColorScheme** function sends the server application the color palette recommended by the client application.

Parameters

lpDoc
Points to an **OLESERVERDOC** structure for which the client application recommends a palette.

lpPal
Points to a **LOGPALETTE** structure specifying the recommended palette.

Return Value

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

Comments

Server applications are not required to use the palette recommended by the client application.

Before returning from the **SetColorScheme** function, the server application should create a handle to the palette. To do this, the server application should use the palette pointed to by the *lpPal* parameter in a call to the **CreatePalette** function, as shown in the following example.

```
hpal = CreatePalette(lpPal);
```

The server can then use the palette handle to refer to the palette.

The first palette entry in the **LOGPALETTE** structure specifies the foreground color recommended by the client application. The second palette entry specifies the background color. The first half of the remaining palette entries are fill colors, and the second half are colors for lines and text.

Client applications typically specify an even number of palette entries. When there is an uneven number of entries, the server should interpret the odd entry as a fill color; that is, if there are five entries, three should be interpreted as fill colors and two as line and text colors.

Function Execute

OLESTATUS Execute(*lpDoc*, *hCommands*)
LPOLESERVERDOC *lpDoc*;
HGLOBAL *hCommands*;

The **Execute** function receives WM_DDE_EXECUTE commands sent by client applications. The applications send these commands by calling the **OleExecute** function.

Parameters

lpDoc
Points to an **OLESERVERDOC** structure to which the dynamic data exchange (DDE) commands apply.

hCommands
Identifies memory containing one or more DDE execute commands.

Return Value

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

Comments

The server should never free the handle specified in the *hCommands* parameter.

OLESERVERVTBL

3.1

```
#include <ole.h>

typedef struct _OLESERVERVTBL {  /* osv */
    OLESTATUS (CALLBACK* Open)(LPOLESERVER, LHSERVERDOC,
        OLE_LPCSTR, LPOLESERVERDOC FAR*);
    OLESTATUS (CALLBACK* Create)(LPOLESERVER, LHSERVERDOC,
        OLE_LPCSTR, OLE_LPCSTR, LPOLESERVERDOC FAR*);
    OLESTATUS (CALLBACK* CreateFromTemplate)(LPOLESERVER,
        LHSERVERDOC, OLE_LPCSTR, OLE_LPCSTR, OLE_LPCSTR,
        LPOLESERVERDOC FAR*);
    OLESTATUS (CALLBACK* Edit)(LPOLESERVER, LHSERVERDOC,
        OLE_LPCSTR, OLE_LPCSTR, LPOLESERVERDOC FAR*);
    OLESTATUS (CALLBACK* Exit)(LPOLESERVER);
    OLESTATUS (CALLBACK* Release)(LPOLESERVER);
    OLESTATUS (CALLBACK* Execute)(LPOLESERVER, HGLOBAL);
} OLESERVERVTBL;
```

The **OLESERVERVTBL** structure points to functions that manipulate a server. After a server application creates this structure and an **OLESERVER** structure, the server library can perform operations on the server application.

Every function except **Release** can return OLE_BUSY.

Function Open

OLESTATUS Open(*lpServer*, *lhDoc*, *lpszDoc*, *lplpDoc*)
LPOLESERVER *lpServer*;
LHSERVERDOC *lhDoc*;
OLE_LPCSTR *lpszDoc*;
LPOLESERVERDOC FAR* *lplpDoc*;

The **Open** function opens an existing file and prepares to edit the contents. A server typically uses this function to open a linked object for a client application.

Parameters

lpServer
Points to an **OLESERVER** structure identifying the server.

lhDoc
Identifies the document. The library uses this handle internally.

lpszDoc
Points to a null-terminated string specifying the permanent name of the document to be opened. Typically this string is a path, but for some applications it might be further qualified. For example, the string might specify a particular table in a database.

lplpDoc
Points to a variable of type **LPOLESERVERDOC** in which the server application returns a long pointer to the **OLESERVERDOC** structure it has created in response to this function.

Return Value

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

Comments

When the library calls this function, the server application opens a specified document, allocates and initializes an **OLESERVERDOC** structure, associates the library's handle with the document, and returns the address of the structure. The server does not show the document or its window.

Function

Create

OLESTATUS Create(*lpServer*, *lhDoc*, *lpszClass*, *lpszDoc*, *lplpDoc*)
LPOLESERVER *lpServer*;
LHSERVERDOC *lhDoc*;
OLE_LPCSTR *lpszClass*;
OLE_LPCSTR *lpszDoc*;
LPOLESERVERDOC FAR* *lplpDoc*;

The **Create** function makes a new object that is to be embedded in the client application. The *lpszDoc* parameter identifies the object but should not be used to create a file for the object.

Parameters

lpServer
Points to an **OLESERVER** structure identifying the server.

lhDoc
Identifies the document. The library uses this handle internally.

lpszClass
Points to a null-terminated string specifying the class of document to create.

lpszDoc
Points to a null-terminated string specifying a name for the document to be created. This name can be used to identify the document in window titles.

lplpDoc
Points to a variable of type **LPOLESERVERDOC** in which the server application should return a long pointer to the created **OLESERVERDOC** structure.

Return Value

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

Comments

When the library calls this function, the server application creates a document of a specified class, allocates and initializes an **OLESERVERDOC** structure, associates the library's handle with the document, and returns the address of the structure. This function opens the created document for editing and embeds it in the client when it is updated or closed.

Server applications often track changes to the document specified in this function, so that the user can be prompted to save changes when necessary.

Function

CreateFromTemplate

OLESTATUS CreateFromTemplate(*lpServer*, *lhDoc*, *lpszClass*, *lpszDoc*, *lpszTemplate*, *lplpDoc*)
LPOLESERVER *lpServer*;
LHSERVERDOC *lhDoc*;
OLE_LPCSTR *lpszClass*;
OLE_LPCSTR *lpszDoc*;
OLE_LPCSTR *lpszTemplate*;
LPOLESERVERDOC FAR* *lplpDoc*;

The **CreateFromTemplate** function creates a new document that is initialized with the data in a specified file. The new document is opened for editing by this function and embedded in the client when it is updated or closed.

Parameters

lpServer
Points to an **OLESERVER** structure identifying the server.

lhDoc
Identifies the document. The library uses this handle internally.

lpszClass
Points to a null-terminated string specifying the class of document to create.

lpszDoc
Points to a null-terminated string specifying a name for the document to be created. This name need not be used by the server application but can be used in window titles.

lpszTemplate
Points to a null-terminated string specifying the permanent name of the document to use to initialize the new document. Typically this string is a path, but for some applications it might be further qualified. For example, the string might specify a particular table in a database.

lplpDoc
Points to a variable of type **LPOLESERVERDOC** in which the server application should return a long pointer to the created **OLESERVERDOC** structure.

Return Value

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

Comments

When the library calls this function, the server application creates a document of a specified class, allocates and initializes an **OLESERVERDOC** structure, associates the library's handle with the document, and returns the address of the structure.

A server application often tracks changes to the document specified in this function, so that the user can be prompted to save changes when necessary.

Function **Edit**

OLESTATUS Edit(*lpServer*, *lhDoc*, *lpszClass*, *lpszDoc*, *lplpDoc*)
LPOLESERVER *lpServer*;
LHSERVERDOC *lhDoc*;
OLE_LPCSTR *lpszClass*;
OLE_LPCSTR *lpszDoc*;
LPOLESERVERDOC FAR* *lplpDoc*;

The **Edit** function creates a document that is initialized with data retrieved by a subsequent call to the **SetData** function. The object is embedded in the client application. The server does not show the document or its window.

Parameters

lpServer
Points to an **OLESERVER** structure identifying the server.

lhDoc
Identifies the document. The library uses this handle internally.

lpszClass
Points to a null-terminated string specifying the class of document to create.

lpszDoc
Points to a null-terminated string specifying a name for the document to be created. This name need not be used by the server application but may be used—for example, in a window title.

lplpDoc
Points to a variable of type LPOLESERVERDOC in which the server application should return a long pointer to the created **OLESERVERDOC** structure.

Return Value

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

Comments

When the library calls this function, the server application creates a document of a specified class, allocates and initializes an **OLESERVERDOC** structure, associates the library's handle with the document, and returns the address of the structure.

The document created by the **Edit** function retrieves the initial data from the client in a subsequent call to the **SetData** function. The user can edit the document after the data has been retrieved and the library has used either the **Show** function in the **OLEOBJECTVTBL** structure or the **DoVerb** function with an Edit verb to show the document to the user.

Function Exit

OLESTATUS Exit(*lpServer*)
LPOLESERVER *lpServer*;

The **Exit** function instructs the server application to close documents and quit.

Parameters

lpServer
Points to an **OLESERVER** structure identifying the server.

Return Value

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

Comments

The server library calls the **Exit** function to instruct a server application to terminate. If the server application has no open documents when the **Exit** function is called, it should call the **OleRevokeServer** function.

Function Release

OLESTATUS Release(*lpServer*)
LPOLESERVER *lpServer*;

The **Release** function notifies a server that all connections to it have closed and that it is safe to quit.

Parameters

lpServer
Points to an **OLESERVER** structure identifying the server.

Return Value

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

Comments

The server library calls the **Release** function when it is safe for a server to quit. When a server application calls the **OleRevokeServer** function, the application must continue to dispatch messages and wait for the library to call the **Release** function before quitting.

When the server is invisible and the library calls **Release**, the server must exit. (The only exception is when an application supports multiple servers; in this case, an invisible server is sometimes not revocable when the library calls **Release**.) If the server has no open documents and it was started with the **/Embedding** option (indicating that it was started by a client application), the server should exit when the library calls the **Release** function. If the user has explicitly loaded a document into a single-instance multiple document interface server, however, the server should not exit when the library calls **Release**. Typically, a single-instance server is a multiple document interface (MDI) server.

All registered server structures must be released before a server can quit.

A server can call the **PostQuitMessage** function inside the **Release** function.

Function

Execute

OLESTATUS Execute(*lpServer***, *hCommands*)**
LPOLESERVER *lpServer*;
HGLOBAL *hCommands*;

The **Execute** function receives WM_DDE_EXECUTE commands sent by client applications. The applications send these commands by calling the **OleExecute** function.

Parameters

lpServer
Points to an **OLESERVER** structure identifying the server.

hCommands
Identifies memory containing one or more dynamic data exchange (DDE) execute commands.

Return Value

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

Comments

The server should never free the handle specified in the *hCommands* parameter.

OLESTREAM

3.1

```
#include <ole.h>

typedef struct _OLESTREAM {      /* ostr */
    LPOLESTREAMVTBL lpstbl;
} OLESTREAM;
```

The **OLESTREAM** structure points to an **OLESTREAMVTBL** structure that provides stream input and output functions. These functions are used by the client library for stream operations on objects. The **OLESTREAM** structure is allocated and initialized by client applications.

Members

lpstbl
Points to an **OLESTREAMVTBL** structure.

OLESTREAMVTBL

3.1

```
#include <ole.h>

typedef struct _OLESTREAMVTBL {    /* ostrv */
    DWORD (CALLBACK* Get)(LPOLESTREAM, void FAR*, DWORD);
    DWORD (CALLBACK* Put)(LPOLESTREAM, OLE_CONST void FAR*, DWORD);
} OLESTREAMVTBL;
```

The **OLESTREAMVTBL** structure points to functions the client library uses for stream operations on objects. This structure is allocated and initialized by client applications.

Comments

The stream is valid only for the duration of the function to which it is passed. The library obtains everything it requires while the stream is valid.

The return values for the stream functions may indicate that an error has occurred, but these values do not indicate the nature of the error. The client application is responsible for any required error-recovery operations.

A client application can use these functions to provide variations on the standard stream procedures; for example, the client could change the permanent storage of some objects so that they were stored in a database instead of the client document.

Function Get

DWORD Get(*lpstream*, *lpszBuf*, *cbbuf*)
LPOLESTREAM *lpstream*;
void FAR* *lpszBuf*;
DWORD *cbbuf*;

The **Get** function gets data from the specified stream.

Parameters

lpstream
Points to an **OLESTREAM** structure allocated by the client.

lpszBuf
Points to a buffer to fill with data from the stream.

cbbuf
Specifies the number of bytes to read into the buffer.

Return Value

The return value is the number of bytes actually read into the buffer if the function is successful. If the end of the file is encountered, the return value is zero. A negative return value indicates that an error occurred.

Comments

The value specified by the *cbbuf* parameter can be larger than 64K. If the client application uses a stream-reading function that is limited to 64K, it should call that function repeatedly until it has read the number of bytes specified by *cbbuf*. Whenever the data size is larger than 64K, the pointer to the data buffer is always at the beginning of the segment.

Function Put

DWORD Put(*lpstream*, *lpszBuf*, *cbbuf*)
LPOLESTREAM *lpstream*;
OLE_CONST void FAR* *lpszBuf*;
DWORD *cbbuf*;

The **Put** function puts data into the specified stream.

Parameters

lpstream
Points to an **OLESTREAM** structure allocated by the client.

lpszBuf
Points to a buffer from which to write data into the stream.

cbbuf
Specifies the number of bytes to write into the stream.

Return Value

The return value is the number of bytes actually written to the stream. A return value less than the number specified in the *cbbuf* parameter indicates that either there was insufficient space in the stream or an error occurred.

Comments

The value specified by the *cbbuf* parameter can be greater than 64K. If the client application uses a stream-writing function that is limited to 64K, it should call that function repeatedly until it has written the number of bytes specified by *cbbuf*. Whenever the data size is greater than 64K, the pointer to the data buffer is always at the beginning of the segment.

OLETARGETDEVICE

3.1

```
#include <ole.h>

typedef struct _OLETARGETDEVICE {
    UINT otdDeviceNameOffset;
    UINT otdDriverNameOffset;
    UINT otdPortNameOffset;
    UINT otdExtDevmodeOffset;
    UINT otdExtDevmodeSize;
    UINT otdEnvironmentOffset;
    UINT otdEnvironmentSize;
    BYTE otdData[1];
} OLETARGETDEVICE;
```

The **OLETARGETDEVICE** structure contains information about the target device that a client application is using. Server applications can use the information in this structure to change the rendering of an object, if necessary. A client application provides a handle to this structure in a call to the **OleSetTargetDevice** function.

Members

otdDeviceNameOffset
Specifies the offset from the beginning of the array to the name of the device.

otdDriverNameOffset
Specifies the offset from the beginning of the array to the name of the device driver.

otdPortNameOffset
Specifies the offset from the beginning of the array to the name of the port.

otdExtDevmodeOffset
Specifies the offset from the beginning of the array to a **DEVMODE** structure retrieved by the **ExtDeviceMode** function.

otdExtDevmodeSize
Specifies the size of the **DEVMODE** structure whose offset is specified by the **otdExtDevmodeOffset** member.

otdEnvironmentOffset
Specifies the offset from the beginning of the array to the device environment.

otdEnvironmentSize
Specifies the size of the environment whose offset is specified by the **otdEnvironmentOffset** member.

otdData
Specifies an array of bytes containing data for the target device.

Comments

The **otdDeviceNameOffset**, **otdDriverNameOffset**, and **otdPortNameOffset** members should be null-terminated.

In Windows 3.1, the ability to connect multiple printers to one port has made the environment obsolete. The environment information retrieved by the **GetEnvironment** function can occasionally be incorrect. To ensure that the **OLETARGETDEVICE** structure is initialized correctly, the application should copy information from the **DEVMODE** structure retrieved by a call to the **ExtDeviceMode** function to the environment position of the **OLETARGETDEVICE** structure.

See Also

OleSetTargetDevice

OPENFILENAME

3.1

```
#include <commdlg.h>

typedef struct tagOPENFILENAME { /* ofn */
    DWORD      lStructSize;
    HWND       hwndOwner;
    HINSTANCE  hInstance;
    LPCSTR     lpstrFilter;
    LPSTR      lpstrCustomFilter;
    DWORD      nMaxCustFilter;
    DWORD      nFilterIndex;
    LPSTR      lpstrFile;
    DWORD      nMaxFile;
    LPSTR      lpstrFileTitle;
    DWORD      nMaxFileTitle;
    LPCSTR     lpstrInitialDir;
    LPCSTR     lpstrTitle;
    DWORD      Flags;
    UINT       nFileOffset;
    UINT       nFileExtension;
    LPCSTR     lpstrDefExt;
    LPARAM     lCustData;
    UINT       (CALLBACK *lpfnHook) (HWND, UINT, WPARAM, LPARAM);
    LPCSTR     lpTemplateName;
} OPENFILENAME;
```

The **OPENFILENAME** structure contains information that the system uses to initialize the system-defined Open dialog box or Save dialog box. After the user chooses the OK button to close the dialog box, the system returns information about the user's selection in this structure.

Members

lStructSize

Specifies the length of the structure, in bytes. This member is filled on input.

hwndOwner

Identifies the window that owns the dialog box. This member can be any valid window handle, or it should be NULL if the dialog box is to have no owner.

If the OFN_SHOWHELP flag is set, **hwndOwner** must identify the window that owns the dialog box. The window procedure for this owner window receives a notification message when the user chooses the Help button. (The identifier for the notification message is the value returned by the **RegisterWindowMessage** function when HELPMSGSTRING is passed as its argument.)

This member is filled on input.

hInstance
Identifies a data block that contains a dialog box template specified by the **lpTemplateName** member. This member is used only if the **Flags** member specifies the OFN_ENABLETEMPLATE or the OFN_ENABLETEMPLATEHANDLE flag; otherwise, this member is ignored.

This member is filled on input.

lpstrFilter
Points to a buffer containing one or more pairs of null-terminated strings specifying filters. The first string in each pair describes a filter (for example, "Text Files"); the second specifies the filter pattern (for example, "*.txt"). Multiple filters can be specified for a single item; in this case, the semicolon (;) is used to separate filter pattern strings—for example, "*.txt;*.doc;*.bak". The last string in the buffer must be terminated by two null characters. If this parameter is NULL, the dialog box does not display any filters. The filter strings must be in the proper order—the system does not change the order.

This member is filled on input.

lpstrCustomFilter
Points to a buffer containing a pair of user-defined strings that specify a filter. The first string describes the filter, and the second specifies the filter pattern (for example, "WinWord", "*.doc"). The buffer is terminated by two null characters. The system copies the strings to the buffer when the user chooses the OK button to close the dialog box. The system uses the strings as the initial filter description and filter pattern for the dialog box. If this parameter is NULL, the dialog box lists (but does not save) user-defined filter strings.

nMaxCustFilter
Specifies the size, in bytes, of the buffer identified by the **lpstrCustomFilter** member. This buffer should be at least 40 bytes long. This parameter is ignored if the **lpstrCustomFilter** member is NULL.

This member is filled on input.

nFilterIndex
Specifies an index into the buffer pointed to by the **lpstrFilter** member. The system uses the index value to obtain a pair of strings to use as the initial filter description and filter pattern for the dialog box. The first pair of strings has an index value of 1. When the user chooses the OK button to close the dialog box, the system copies the index of the selected filter strings into this location.
If the **nFilterIndex** member is 0, the filter in the buffer pointed to by the **lpstrCustomFilter** member is used. If the **nFilterIndex** member is 0 and the **lpstrCustomFilter** member is NULL, the system uses the first filter in the buffer pointed to by the **lpstrFilter** member. If each of the three members is either 0 or NULL, the system does not use any filters and does not show any files in the File Name list box of the dialog box.

lpstrFile
Points to a buffer that specifies a filename used to initialize the File Name edit control. If initialization is not necessary, the first character of this buffer must be NULL. When the **GetOpenFileName** or **GetSaveFileName** function returns, this buffer contains the complete location and name of the selected file.

If the buffer is too small, the dialog box procedure copies the required size into this member and returns 0. To retrieve the required size, cast the **lpstrFile** member to type **LPWORD**. The buffer must be at least three bytes to receive the required size. When the buffer is too small, the **CommDlgExtendedError** function returns the FNERR_BUFFERTOOSMALL value.

nMaxFile
Specifies the size, in bytes, of the buffer pointed to by the **lpstrFile** member. The **GetOpenFileName** and **GetSaveFileName** functions return FALSE if the buffer is too small to contain the file information. The buffer should be at least 256 bytes long. If the **lpstrFile** member is NULL, this member is ignored.

This member is filled on input.

lpstrFileTitle
Points to a buffer that receives the title of the selected file. This buffer receives the filename and extension but no path information. An application should use this string to display the file title. If this member is NULL, the function does not copy the file title. This member is filled on output.

nMaxFileTitle
Specifies the maximum length, in bytes, of the string that can be copied into the **lpstrFileTitle** buffer. This member is ignored if **lpstrFileTitle** is NULL. This member is filled on input.

lpstrInitialDir
Points to a string that specifies the initial file directory. If this member is NULL, the system uses the current directory as the initial directory. (If the **lpstrFile** member contains a string that specifies a valid path, the common dialog box procedure will use the path specified by this string *instead of* the path specified by the string to which **lpstrInitialDir** points.)

This member is filled on input.

lpstrTitle
Points to a string to be placed in the title bar of the dialog box. If this member is NULL, the system uses the default title (that is, Save As or Open). This member is filled on input.

Flags
Specifies the dialog box initialization flags. This member may be a combination of the following values:

Value	Meaning
OFN_ALLOWMULTISELECT	Specifies that the File Name list box is to allow multiple selections. When this flag is set, the **lpstrFile** member points to a buffer containing the path to the current directory and all filenames in the selection. The first filename is separated from the path by a space. Each subsequent filename is separated by one space from the preceding filename. Some of the selected filenames may be preceded by relative paths; for example, the buffer could contain something like this: c:\files file1.txt file2.txt ..\bin\file3.txt
OFN_CREATEPROMPT	Causes the dialog box procedure to generate a message box to notify the user when a specified file does not currently exist and to make it possible for the user to specify that the file should be created. (This flag automatically sets the OFN_PATHMUSTEXIST and OFN_FILEMUSTEXIST flags.)
OFN_ENABLEHOOK	Enables the hook function specified in the **lpfnHook** member.
OFN_ENABLETEMPLATE	Causes the system to use the dialog box template identified by the **hInstance** and **lpTemplateName** members to create the dialog box.
OFN_ENABLETEMPLATEHANDLE	Indicates that the **hInstance** member identifies a data block that contains a pre-loaded dialog box template. The system ignores the **lpTemplateName** member if this flag is specified.
OFN_EXTENSIONDIFFERENT	Indicates that the extension of the returned filename is different from the extension specified by the **lpstrDefExt** member. This flag is not set if **lpstrDefExt** is NULL, if the extensions match, or if the file has no extension. This flag can be set on output.
OFN_FILEMUSTEXIST	Specifies that the user can type only the names of existing files in the File Name edit control. If this flag is set and the user types an invalid filename in the File Name edit control, the dialog box procedure displays a warning in a message box. (This flag also causes the OFN_PATHMUSTEXIST flag to be set.)
OFN_HIDEREADONLY	Hides the Read Only check box.

Value	Meaning
OFN_NOCHANGEDIR	Forces the dialog box to reset the current directory to what it was when the dialog box was created.
OFN_NOREADONLYRETURN	Specifies that the file returned will not have the Read Only attribute set and will not be in a write-protected directory.
OFN_NOTESTFILECREATE	Specifies that the file will not be created before the dialog box is closed. This flag should be set if the application saves the file on a create-no-modify network share point. When an application sets this flag, the library does not check against write protection, a full disk, an open drive door, or network protection. Therefore, applications that use this flag must perform file operations carefully—a file cannot be reopened once it is closed.
OFN_NOVALIDATE	Specifies that the common dialog boxes will allow invalid characters in the returned filename. Typically, the calling application uses a hook function that checks the filename using the FILEOKSTRING registered message. If the text in the edit control is empty or contains nothing but spaces, the lists of files and directories are updated. If the text in the edit control contains anything else, the **nFileOffset** and **nFileExtension** members are set to values generated by parsing the text. No default extension is added to the text, nor is text copied to the **lpstrFileTitle** buffer. If the value specified by the **nFileOffset** member is negative, the filename is invalid. If the value specified by **nFileOffset** is not negative, the filename is valid, and **nFileOffset** and **nFileExtension** can be used as if the OFN_NOVALIDATE flag had not been set.
OFN_OVERWRITEPROMPT	Causes the Save As dialog box to generate a message box if the selected file already exists. The user must confirm whether to overwrite the file.
OFN_PATHMUSTEXIST	Specifies that the user can type only valid paths. If this flag is set and the user types an invalid path in the File Name edit control, the dialog box procedure displays a warning in a message box.
OFN_READONLY	Causes the Read Only check box to be initially checked when the dialog box is created. When the user chooses the OK button to close the dialog box, the state of the Read Only check box is specified by this member. This flag can be set on input and output.

Value	Meaning
OFN_SHAREAWARE	Specifies that if a call to the **OpenFile** function has failed because of a network sharing violation, the error is ignored and the dialog box returns the given filename. If this flag is not set, the registered message for SHAREVISTRING is sent to the hook function, with a pointer to a null-terminated string for the path name in the *lParam* parameter. The hook function responds with one of the following values:

Value	Meaning
OFN_SHAREFALLTHROUGH	Specifies that the filename is returned from the dialog box.
OFN_SHARENOWARN	Specifies no further action.
OFN_SHAREWARN	Specifies that the user receives the standard warning message for this error. (This is the same result as if there were no hook function.)

This flag may be set on output.

OFN_SHOWHELP	Causes the dialog box to show the Help push button. The **hwndOwner** must not be NULL if this option is specified.

These flags may be set when the structure is initialized, except where specified.

nFileOffset

Specifies a zero-based offset from the beginning of the path to the filename specified by the string in the buffer to which **lpstrFile** points. For example, if **lpstrFile** points to the string, "c:\dir1\dir2\file.ext", this member contains the value 13.

This member is filled on output.

nFileExtension

Specifies a zero-based offset from the beginning of the path to the filename extension specified by the string in the buffer to which **lpstrFile** points. For example, if **lpstrFile** points to the following string, "c:\dir1\dir2\file.ext", this member contains the value 18. If the user did not type an extension *and* **lpstrDefExt** is NULL, this member specifies an offset to the terminating null character. If the user typed a period (.) as the last character in the filename, this member is 0.

This member is filled on output.

lpstrDefExt

Points to a buffer that contains the default extension. The **GetOpenFileName** or **GetSaveFileName** function appends this extension to the filename if the user fails to enter an extension. If the filename with the default extension is not found, **GetOpenFileName** or **GetSaveFileName** attempts to find the file by using the name exactly as the user typed it. This string can be any length, but only the first three characters are appended. The string should *not* contain a period (.). If this member is NULL and the user fails to type an extension, no extension is appended. This member is filled on input.

lCustData

Specifies application-defined data that the system passes to the hook function pointed to by the **lpfnHook** member. The system passes a pointer to the **OPENFILENAME** structure in the *lParam* parameter of the WM_INITDIALOG message; this pointer can be used to retrieve the **lCustData** member.

lpfnHook

Points to a hook function that processes messages intended for the dialog box. To enable the hook function, an application must specify the OFN_ENABLEHOOK flag in the **Flags** member; otherwise, the system ignores this structure member. The hook function must return zero to pass a message that it didn't process back to the dialog box procedure in COMMDLG.DLL. The hook function must return a nonzero value to prevent the dialog box procedure in COMMDLG.DLL from processing a message it has already processed.

This member is filled on input.

lpTemplateName

Points to a null-terminated string that specifies the name of the resource file for the dialog box template that is to be substituted for the dialog box template in COMMDLG.DLL. An application can use the **MAKEINTRESOURCE** macro for numbered dialog box resources. This member is used only if the **Flags** member specifies the OFN_ENABLETEMPLATE flag; otherwise, this member is ignored.

This member is filled on input.

See Also **GetOpenFileName, GetSaveFileName**

OUTLINETEXTMETRIC

3.1

```
typedef struct tagOUTLINETEXTMETRIC {
    UINT        otmSize;
    TEXTMETRIC  otmTextMetrics;
    BYTE        otmFiller;
    PANOSE      otmPanoseNumber;
    UINT        otmfsSelection;
    UINT        otmfsType;
    UINT        otmsCharSlopeRise;
    UINT        otmsCharSlopeRun;
    UINT        otmItalicAngle;
    UINT        otmEMSquare;
    INT         otmAscent;
    INT         otmDescent;
    UINT        otmLineGap;
    UINT        otmsXHeight;
    UINT        otmsCapEmHeight;
    RECT        otmrcFontBox;
    INT         otmMacAscent;
    INT         otmMacDescent;
    UINT        otmMacLineGap;
    UINT        otmusMinimumPPEM;
    POINT       otmptSubscriptSize;
    POINT       otmptSubscriptOffset;
    POINT       otmptSuperscriptSize;
    POINT       otmptSuperscriptOffset;
    UINT        otmsStrikeoutSize;
    INT         otmsStrikeoutPosition;
    INT         otmsUnderscorePosition;
    UINT        otmsUnderscoreSize;
    PSTR        otmpFamilyName;
    PSTR        otmpFaceName;
    PSTR        otmpStyleName;
    PSTR        otmpFullName;
} OUTLINETEXTMETRIC;
```

The **OUTLINETEXTMETRIC** structure contains metrics describing a TrueType font.

Members

otmSize
Specifies the size, in bytes, of the **OUTLINETEXTMETRIC** structure.

otmTextMetrics
Specifies a **TEXTMETRIC** structure containing further information about the font.

otmFiller
Specifies a value that causes the structure to be byte-aligned.

otmPanoseNumber
Specifies the Panose number for this font.

otmfsSelection
Specifies the nature of the font pattern. This member can be a combination of the following bits:

Bit	Meaning
0	Italic
1	Underscore
2	Negative
3	Outline
4	Strikeout
5	Bold

otmfsType
Specifies whether the font is licensed. Licensed fonts may not be modified or exchanged. If bit 1 is set, the font may not be embedded in a document. If bit 1 is clear, the font can be embedded. If bit 2 is set, the embedding is read-only.

otmsCharSlopeRise
Specifies the slope of the cursor. This value is 1 if the slope is vertical. Applications can use this value and the value of the **otmsCharSlopeRun** member to create an italic cursor that has the same slope as the main italic angle (specified by the **otmItalicAngle** member).

otmsCharSlopeRun
Specifies the slope of the cursor. This value is zero if the slope is vertical. Applications can use this value and the value of the **otmsCharSlopeRise** member to create an italic cursor that has the same slope as the main italic angle (specified by the **otmItalicAngle** member).

otmItalicAngle
Specifies the main italic angle of the font, in counterclockwise degrees from vertical. Regular (roman) fonts have a value of zero. Italic fonts typically have a negative italic angle (that is, they lean to the right).

otmEMSquare
Specifies the number of logical units defining the x- or y-dimension of the em square for this font. (The number of units in the x- and y-directions are always the same for an em square.)

otmAscent
Specifies the maximum distance characters in this font extend above the base line. This is the typographic ascent for the font.

otmDescent
Specifies the maximum distance characters in this font extend below the base line. This is the typographic descent for the font.

otmLineGap
Specifies typographic line spacing.

otmsXHeight
Not supported.

otmsCapEmHeight
Not supported.

otmrcFontBox
Specifies the bounding box for the font.

otmMacAscent
Specifies the maximum distance characters in this font extend above the base line for the Macintosh.

otmMacDescent
Specifies the maximum distance characters in this font extend below the base line for the Macintosh.

otmMacLineGap
Specifies line-spacing information for the Macintosh.

otmusMinimumPPEM
Specifies the smallest recommended size for this font, in pixels per em-square.

otmptSubscriptSize
Specifies the recommended horizontal and vertical size for subscripts in this font.

otmptSubscriptOffset
Specifies the recommended horizontal and vertical offset for subscripts in this font. The subscript offset is measured from the character origin to the origin of the subscript character.

otmptSuperscriptSize
Specifies the recommended horizontal and vertical size for superscripts in this font.

otmptSuperscriptOffset
Specifies the recommended horizontal and vertical offset for superscripts in this font. The subscript offset is measured from the character base line to the base line of the superscript character.

otmsStrikeoutSize
Specifies the width of the strikeout stroke for this font. Typically, this is the width of the em-dash for the font.

otmsStrikeoutPosition
Specifies the position of the strikeout stroke relative to the base line for this font. Positive values are above the base line and negative values are below.

otmsUnderscorePosition
Specifies the position of the underscore character for this font.

otmsUnderscoreSize
Specifies the thickness of the underscore character for this font.

otmpFamilyName
Specifies the offset from the beginning of the structure to a string specifying the family name for the font.

otmpFaceName
Specifies the offset from the beginning of the structure to a string specifying the face name for the font. (This face name corresponds to the name specified in the **LOGFONT** structure.)

otmpStyleName
Specifies the offset from the beginning of the structure to a string specifying the style name for the font.

otmpFullName
Specifies the offset from the beginning of the structure to a string specifying the full name for the font. This name is unique for the font and often contains a version number or other identifying information.

Comments

The sizes returned in **OUTLINETEXTMETRIC** are given in logical units; that is, they depend on the current mapping mode of the specified display context.

See Also

GetOutlineTextMetrics

PAINTSTRUCT

2.x

```
typedef struct tagPAINTSTRUCT {     /* ps */
   HDC  hdc;
   BOOL fErase;
   RECT rcPaint;
   BOOL fRestore;
   BOOL fIncUpdate;
   BYTE rgbReserved[16];
} PAINTSTRUCT;
```

The **PAINTSTRUCT** structure contains information for an application. This information can be used to paint the client area of a window owned by that application.

Members

hdc
Identifies the display context to be used for painting.

fErase
Specifies whether the background needs to be redrawn. This value is nonzero if the application should redraw the background. The application is responsible

for drawing the background if a window class is created without a background brush. For more information, see the description of the **hbrBackground** member of the **WNDCLASS** structure.

rcPaint
Specifies the upper-left and lower-right corners of the rectangle in which the painting is requested.

fRestore
Reserved; used internally by Windows.

fIncUpdate
Reserved; used internally by Windows.

rgbReserved
Reserved (reserved memory object used internally by Windows).

See Also **BeginPaint**, **WNDCLASS**

PALETTEENTRY

3.0

```
typedef struct tagPALETTEENTRY {    /* pe */
    BYTE  peRed;
    BYTE  peGreen;
    BYTE  peBlue;
    BYTE  peFlags;
} PALETTEENTRY;
```

The **PALETTEENTRY** structure specifies the color and usage of an entry in a logical color palette. A logical palette is defined by a **LOGPALETTE** structure.

Members

peRed
Specifies the intensity of red for the palette entry color.

peGreen
Specifies the intensity of green for the palette entry color.

peBlue
Specifies the intensity of blue for the palette entry color.

peFlags
Specifies how the palette entry is to be used. The **peFlags** member may be set to NULL or to one of the following values (specifying NULL informs Windows that the palette entry contains an RGB value and that it should be mapped normally):

Value	Meaning
PC_EXPLICIT	Specifies that the low-order word of the logical palette entry designates a hardware palette index. This flag allows the application to show the contents of the palette for the display device.
PC_NOCOLLAPSE	Specifies that the color will be placed in an unused entry in the system palette instead of being matched to an existing color in the system palette. Once this color is in the system palette, colors in other logical palettes can be matched to this color. If there are no unused entries in the system palette, the color is matched normally.
PC_RESERVED	Specifies that the logical palette entry will be used for palette animation. Because the color will frequently change, using this flag prevents other windows from matching colors to this palette entry. If an unused system-palette entry is available, this color is placed in that entry. Otherwise, the color will not be available for animation.

See Also **AnimatePalette**

PANOSE

3.1

```
typedef struct tagPANOSE {    /* panose */
    BYTE bFamilyType;
    BYTE bSerifStyle;
    BYTE bWeight;
    BYTE bProportion;
    BYTE bContrast;
    BYTE bStrokeVariation;
    BYTE bArmStyle;
    BYTE bLetterform;
    BYTE bMidline;
    BYTE bXHeight;
} PANOSE;
```

The **PANOSE** structure describes the Panose font-classification values for a TrueType font.

Members

bFamilyType

Specifies the font family. This member can be one of the following values:

Value	Meaning
0	Any
1	No fit
2	Text and display
3	Script
4	Decorative
5	Pictorial

bSerifStyle

Specifies the style of serifs for the font. This member can be one of the following values:

Value	Meaning
0	Any
1	No fit
2	Cove
3	Obtuse cove
4	Square cove
5	Obtuse square cove
6	Square
7	Thin
8	Bone
9	Exaggerated
10	Triangle
11	Normal sans
12	Obtuse sans
13	Perp sans
14	Flared
15	Rounded

bWeight

Specifies the weight of the font. This member can be one of the following values:

Value	Meaning
0	Any
1	No fit
2	Very light
3	Light
4	Thin
5	Book
6	Medium
7	Demi
8	Bold
9	Heavy
10	Black
11	Nord

bProportion

Specifies the proportion of the font. This member can be one of the following values:

Value	Meaning
0	Any
1	No fit
2	Old style
3	Modern
4	Even width
5	Expanded
6	Condensed
7	Very expanded
8	Very condensed
9	Monospaced

bContrast

Specifies the contrast of the font. This member can be one of the following values:

Value	Meaning
0	Any
1	No fit
2	None
3	Very low
4	Low
5	Medium low
6	Medium
7	Medium high
8	High
9	Very high

bStrokeVariation

Specifies the stroke variation for the font. This member can be one of the following values:

Value	Meaning
0	Any
1	No fit
2	Gradual/diagonal
3	Gradual/transitional
4	Gradual/vertical
5	Gradual/horizontal
6	Rapid/vertical
7	Rapid/horizontal
8	Instant/vertical

bArmStyle

Specifies the style for the arms in the font. This member can be one of the following values:

Value	Meaning
0	Any
1	No fit
2	Straight arms/horizontal
3	Straight arms/wedge
4	Straight arms/vertical
5	Straight arms/single serif
6	Straight arms/double serif
7	Non-straight arms/horizontal
8	Non-straight arms/wedge
9	Non-straight arms/vertical
10	Non-straight arms/single serif
11	Non-straight arms/double serif

bLetterform

Specifies the letter form for the font. This member can be one of the following values:

Value	Meaning
0	Any
1	No fit
2	Normal/contact
3	Normal/weighted
4	Normal/boxed
5	Normal/flattened
6	Normal/rounded
7	Normal/off-center
8	Normal/square
9	Oblique/contact
10	Oblique/weighted
11	Oblique/boxed
12	Oblique/flattened
13	Oblique/rounded
14	Oblique/off-center
15	Oblique/square

bMidline

Specifies the style of the midline for the font. This member can be one of the following values:

Value	Meaning
0	Any
1	No fit
2	Standard/trimmed
3	Standard/pointed
4	Standard/serifed
5	High/trimmed
6	High/pointed
7	High/serifed
8	Constant/trimmed
9	Constant/pointed
10	Constant/serifed
11	Low/trimmed
12	Low/pointed
13	Low/serifed

bXHeight

Specifies the x-height of the font. This member can be one of the following values:

Value	Meaning
0	Any
1	No fit
2	Constant/small
3	Constant/standard
4	Constant/large
5	Ducking/small
6	Ducking/standard
7	Ducking/large

POINT

2.x

```
typedef struct tagPOINT {    /* pt */
   int x;
   int y;
} POINT;
```

The **POINT** structure defines the x- and y-coordinates of a point.

Members

x
Specifies the x-coordinate of a point.

y
Specifies the y-coordinate of a point.

See Also **ChildWindowFromPoint, PtInRect, WindowFromPoint**

POINTFX

3.1

```
typedef struct tagPOINTFX {
    FIXED x;
    FIXED y;
} POINTFX;
```

The **POINTFX** structure contains the coordinates of points that describe the outline of a character in a TrueType font. **POINTFX** is a member of the **TTPOLYCURVE** and **TTPOLYGONHEADER** structures.

Members

x
Specifies the x-component of a point on the outline of a TrueType character.

y
Specifies the y-component of a point on the outline of a TrueType character.

See Also **FIXED, TTPOLYCURVE, TTPOLYGONHEADER**

PRINTDLG

3.1

```
#include <commdlg.h>

typedef struct tagPD {  /* pd */
    DWORD     lStructSize;
    HWND      hwndOwner;
    HGLOBAL   hDevMode;
    HGLOBAL   hDevNames;
    HDC       hDC;
    DWORD     Flags;
    UINT      nFromPage;
    UINT      nToPage;
    UINT      nMinPage;
    UINT      nMaxPage;
    UINT      nCopies;
    HINSTANCE hInstance;
    LPARAM    lCustData;
    UINT    (CALLBACK* lpfnPrintHook)(HWND, UINT, WPARAM, LPARAM);
    UINT    (CALLBACK* lpfnSetupHook)(HWND, UINT, WPARAM, LPARAM);
    LPCSTR    lpPrintTemplateName;
    LPCSTR    lpSetupTemplateName;
    HGLOBAL   hPrintTemplate;
    HGLOBAL   hSetupTemplate;
} PRINTDLG;
```

The **PRINTDLG** structure contains information that the system uses to initialize the system-defined Print dialog box. After the user chooses the OK button to close the dialog box, the system returns information about the user's selections in this structure.

Members

lStructSize

Specifies the length of the structure, in bytes. This member is filled on input.

hwndOwner

Identifies the window that owns the dialog box. This member can be any valid window handle, or it should be NULL if the dialog box is to have no owner.

If the PD_SHOWHELP flag is set, **hwndOwner** must identify the window that owns the dialog box. The window procedure for this owner window receives a notification message when the user chooses the Help button. (The identifier for the notification message is the value returned by the **RegisterWindowMessage** function when HELPMSGSTRING is passed as its argument.)

This member is filled on input.

hDevMode

Identifies a movable global memory object that contains a **DEVMODE** structure. Before the **PrintDlg** function is called, the members in this structure may contain data used to initialize the dialog box controls. When the **PrintDlg** function returns, the members in this structure specify the state of each of the dialog box controls.

If the application uses the structure to initialize the dialog box controls, it must allocate space for and create the **DEVMODE** structure. (The application should allocate a movable memory object.)

If the application does not use the structure to initialize the dialog box controls, the **hDevMode** member may be NULL. In this case, the **PrintDlg** function allocates memory for the structure, initializes its members, and returns a handle that identifies it.

If the device driver for the specified printer does not support extended device modes, the **hDevMode** member is NULL when **PrintDlg** returns.

If the device name (specified by the **dmDeviceName** member of the **DEVMODE** structure) does not appear in the [devices] section of WIN.INI, the **PrintDlg** function returns an error.

The value of **hDevMode** may change during the execution of the **PrintDlg** function. This member is filled on input and output.

hDevNames

Identifies a movable global memory object that contains a **DEVNAMES** structure. This structure contains three strings; these strings specify the driver name, the printer name, and the output-port name. Before the **PrintDlg** function is called, the members of this structure contain strings used to initialize the dialog box controls. When the **PrintDlg** function returns, the members of this structure contain the strings typed by the user. The calling application uses these strings to create a device context or an information context.

If the application uses the structure to initialize the dialog box controls, it must allocate space for and create the **DEVMODE** data structure. (The application should allocate a movable global memory object.)

If the application does not use the structure to initialize the dialog box controls, the **hDevNames** member can be NULL. In this case, the **PrintDlg** function allocates memory for the structure, initializes its members (using the printer name specified in the **DEVMODE** data structure), and returns a handle that identifies it. When the **PrintDlg** function initializes the members of the **DEVNAMES** structure, it uses the first port name that appears in the [devices] section of WIN.INI. For example, the function uses "LPT1" as the port name if the following string appears in the [devices] section:

```
PCL / HP LaserJet=HPPCL,LPT1:,LPT2:
```

If both the **hDevMode** and **hDevNames** members are NULL, **PrintDlg** specifies the current default printer for **hDevNames**.

The value of **hDevNames** may change during the execution of the **PrintDlg** function. This member is filled on input and output.

hDC

Identifies either a device context or an information context, depending on whether the **Flags** member specifies the PD_RETURNDC or the PD_RETURNIC flag. If neither flag is specified, the value of this member is undefined. If both flags are specified, **hDC** is PD_RETURNDC.

This member is filled on output.

Flags

Specifies the dialog box initialization flags. This member may be a combination of the following values:

Value	Meaning
PD_ALLPAGES	Indicates that the All radio button was selected when the user closed the dialog box. (This value is used as a placeholder, to indicate that the PD_PAGENUMS and PD_SELECTION flags are not set. This value can be set on input and output.)
PD_COLLATE	Causes the Collate Copies check box to be checked when the dialog box is created. When the **PrintDlg** function returns, this flag indicates the state in which the user left the Collate Copies check box. This flag can be set on input and output.
PD_DISABLEPRINTTOFILE	Disables the Print to File check box.
PD_ENABLEPRINTHOOK	Enables the hook function specified in the **lpfnPrintHook** member of this structure.
PD_ENABLEPRINTTEMPLATE	Causes the system to use the dialog box template identified by the **hInstance** and **lpPrintTemplateName** members to create the Print dialog box.
PD_ENABLEPRINTTEMPLATEHANDLE	Indicates that the **hPrintTemplate** member identifies a data block that contains a pre-loaded dialog box template. The system ignores the **hInstance** member if this flag is specified.
PD_ENABLESETUPHOOK	Enables the hook function specified in the **lpfnSetupHook** member of this structure.
PD_ENABLESETUPTEMPLATE	Causes the system to use the dialog box template identified by the **hInstance** and **lpSetupTemplateName** members to create the Print Setup dialog box.

Value	Meaning
PD_ENABLESETUPTEMPLATEHANDLE	Indicates that the **hSetupTemplate** member identifies a data block that contains a pre-loaded dialog box template. The system ignores the **hInstance** member if this flag is specified.
PD_HIDEPRINTTOFILE	Hides and disables the Print to File check box.
PD_NOPAGENUMS	Disables the Pages radio button and the associated edit controls.
PD_NOSELECTION	Disables the Selection radio button.
PD_NOWARNING	Prevents the warning message from being displayed when there is no default printer.
PD_PAGENUMS	Causes the Pages radio button to be selected when the dialog box is created. When the **PrintDlg** function returns, this flag is set if the Pages button is in the selected state. If neither PD_PAGENUMS nor PD_SELECTION is specified, the All radio button is in the selected state. This flag can be set on input and output.
PD_PRINTSETUP	Causes the system to display the Print Setup dialog box rather than the Print dialog box.
PD_PRINTTOFILE	Causes the Print to File check box to be checked when the dialog box is created. This flag can be set on input and output.
PD_RETURNDC	Causes the **PrintDlg** function to return a device context matching the selections that the user made in the dialog box. The handle to the device context is returned in the **hDC** member. If neither PD_RETURNDC nor PD_RETURNIC is specified, the **hDC** parameter is undefined on output.
PD_RETURNDEFAULT	Causes the **PrintDlg** function to return **DEVMODE** and **DEVNAMES** structures that are initialized for the system default printer. **PrintDlg** does this without displaying a dialog box. Both the **hDevNames** and the **hDevMode** members should be NULL; otherwise, the function returns an error. If the system default printer is supported by an old printer driver (earlier than Windows version 3.0), only the **hDevNames** member is returned—the **hDevMode** member is NULL.

Value	Meaning
PD_RETURNIC	Causes the **PrintDlg** function to return an information context matching the selections that the user made in the dialog box. The information context is returned in the **hDC** member. If neither PD_RETURNDC nor PD_RETURNIC is specified, the **hDC** parameter is undefined on output.
PD_SELECTION	Causes the Selection radio button to be selected when the dialog box is created. When the **PrintDlg** function returns, this flag is set if the Selection button is in the selected state. If neither PD_PAGENUMS nor PD_SELECTION is specified, the All radio button is in the selected state. This flag can be set on input and output.
PD_SHOWHELP	Causes the dialog box to show the Help button. If this flag is specified, the **hwndOwner** must not be NULL.
PD_USEDEVMODECOPIES	Disables the Copies edit control if a printer driver does not support multiple copies. If a driver does support multiple copies, setting this flag indicates that the **PrintDlg** function should store the requested number of copies in the **dmCopies** member of the **DEVMODE** structure and store the value 1 in the **nCopies** member of the **PRINTDLG** structure. If this flag is not set, the **PRINTDLG** structure stores the value 1 in the **dmCopies** member of the **DEVMODE** structure and stores the requested number of copies in the **nCopies** member of the **PRINTDLG** structure.

These flags may be set when the structure is initialized, except where specified.

nFromPage

Specifies the initial value for the starting page in the From edit control. When the **PrintDlg** function returns, this member specifies the page at which to begin printing. This value is valid only if the PD_PAGENUMS flag is specified. The maximum value for this member is 0xFFFE; if 0xFFFF is specified, the From edit control is left blank.

This member is filled on input and output.

nToPage
Specifies the initial value for the ending page in the To edit control. When the **PrintDlg** function returns, this member specifies the last page to print. This value is valid only if the PD_PAGENUMS flag is specified. The maximum value for this member is 0xFFFE; if 0xFFFF is specified, the To edit control is left blank.

This member is filled on input and output.

nMinPage
Specifies the minimum number of pages that can be specified in the From and To edit controls. This member is filled on input.

nMaxPage
Specifies the maximum number of pages that can be specified in the From and To edit controls. This member is filled on input.

nCopies
Before the **PrintDlg** function is called, this member specifies the value to be used to initialize the Copies edit control *if* the **hDevMode** member is NULL; otherwise, the **dmCopies** member of the **DEVMODE** structure contains the value used to initialize the Copies edit control.

When **PrintDlg** returns, the value specified by this member depends on the version of Windows for which the printer driver was written. For printer drivers written for Windows versions earlier than 3.0, this member specifies the number of copies requested by the user in the Copies edit control. For printer drivers written for Windows versions 3.0 and later, this member specifies the number of copies requested by the user *if* the PD_USEDEVMODECOPIES flag was not set; otherwise, this member specifies the value 1 and the actual number of copies requested appears in the **DEVMODE** structure.

This member is filled on input and output.

hInstance
Identifies a data block that contains the pre-loaded dialog box template specified by the **lpPrintTemplateName** or the **lpSetupTemplateName** member. This member is used only if the **Flags** member specifies the PD_ENABLEPRINTTEMPLATE or PD_ENABLESETUPTEMPLATE flag; otherwise, this member is ignored.

This member is filled on input.

lCustData
Specifies application-defined data that the system passes to the hook function identified by the **lpfnPrintHook** or the **lpfnSetupHook** member. The system passes a pointer to the **PRINTDLG** structure in the *lParam* parameter of the WM_INITDIALOG message; this pointer can be used to retrieve the **lCustData** member.

lpfnPrintHook
Points to the exported hook function that processes dialog box messages if the application customizes the Print dialog box. This member is ignored unless the PD_ENABLEPRINTHOOK flag is specified in the **Flags** member.

This member is filled on input.

lpfnSetupHook
Points to the exported hook function that processes dialog box messages if the application customizes the Print Setup dialog box. This member is ignored unless the PD_ENABLESETUPHOOK flag is specified in the **Flags** member.

This member is filled on input.

lpPrintTemplateName
Points to a null-terminated string that specifies the dialog box template that is to be substituted for the standard dialog box template in COMMDLG. An application must specify the PD_ENABLEPRINTTEMPLATE constant in the **Flags** member to enable the hook function; otherwise, the system ignores this structure member.

This member is filled on input.

lpSetupTemplateName
Points to a null-terminated string that specifies the dialog box template that is to be substituted for the standard dialog box template in COMMDLG. An application must specify the PD_ENABLEPRINTTEMPLATE constant in the **Flags** member to enable the hook function; otherwise, the system ignores this structure member.

This member is filled on input.

hPrintTemplate
Identifies the handle of the global memory object that contains the preloaded dialog box template to be used instead of the default template in COMMDLG.DLL for the Print dialog box. To use the dialog box template, the PD_ENABLEPRINTTEMPLATEHANDLE flag must be set.

This member is filled on input.

hSetupTemplate
Identifies the handle of the global memory object that contains the preloaded dialog box template to be used instead of the default template in COMMDLG.DLL for the Print Setup dialog box. To use the dialog box template, the PD_ENABLEPRINTTEMPLATEHANDLE flag must be set.

This member is filled on input.

See Also **CreateDC, CreateIC, PrintDlg, DEVMODE, DEVNAMES**

RASTERIZER_STATUS

3.1

```
typedef struct  tagRASTERIZER_STATUS {    /* rs */
    int   nSize;
    int   wFlags;
    int   nLanguageID;
} RASTERIZER_STATUS;
```

The **RASTERIZER_STATUS** structure contains information about whether TrueType is installed. This structure is filled when an application calls the **GetRasterizerCaps** function.

Members

nSize
Specifies the size, in bytes, of the **RASTERIZER_STATUS** structure.

wFlags
Specifies whether at least one TrueType font is installed and whether TrueType is enabled. This value is TT_AVAILABLE and/or TT_ENABLED if TrueType is on the system.

nLanguageID
Specifies the language in the system's SETUP.INF file. For more information about Microsoft language identifiers, see the **StringTable** structure.

See Also

GetRasterizerCaps

RECT

2.x

```
typedef struct tagRECT {    /* rc */
   int left;
   int top;
   int right;
   int bottom;
} RECT;
```

The **RECT** structure defines the coordinates of the upper-left and lower-right corners of a rectangle.

Members

left
Specifies the x-coordinate of the upper-left corner of a rectangle.

top
Specifies the y-coordinate of the upper-left corner of a rectangle.

right
Specifies the x-coordinate of the lower-right corner of a rectangle.

bottom
Specifies the y-coordinate of the lower-right corner of a rectangle.

Comments

The width of the rectangle defined by the **RECT** structure must not exceed 32,767 units.

When the **RECT** structure is passed to the **FillRect** function, graphics device interface (GDI) fills the rectangle up to, but not including, the right column and bottom row of pixels.

RGBQUAD

3.0

```
typedef struct tagRGBQUAD {     /* rgbq */
    BYTE    rgbBlue;
    BYTE    rgbGreen;
    BYTE    rgbRed;
    BYTE    rgbReserved;
} RGBQUAD;
```

The **RGBQUAD** structure describes a color consisting of relative intensities of red, green, and blue. The **bmiColors** member of the **BITMAPINFO** structure consists of an array of **RGBQUAD** structures.

Members

rgbBlue
Specifies the intensity of blue in the color.

rgbGreen
Specifies the intensity of green in the color.

rgbRed
Specifies the intensity of red in the color.

rgbReserved
Not used; must be set to zero.

RGBTRIPLE

3.0

```
typedef struct tagRGBTRIPLE {    /* rgbt */
    BYTE    rgbtBlue;
    BYTE    rgbtGreen;
    BYTE    rgbtRed;
} RGBTRIPLE;
```

The **RGBTRIPLE** structure describes a color consisting of relative intensities of red, green, and blue. The **bmciColors** member of the **BITMAPCOREINFO** structure consists of an array of **RGBTRIPLE** structures.

Windows applications should use the **BITMAPINFO** structure instead of **BITMAPCOREINFO** whenever possible. The **BITMAPINFO** structure uses an **RGBQUAD** structure instead of the **RGBTRIPLE** structure.

Members

rgbtBlue
Specifies the intensity of blue in the color.

rgbtGreen
Specifies the intensity of green in the color.

rgbtRed
Specifies the intensity of red in the color.

SEGINFO

3.1

```
typedef struct tagSEGINFO {
    UINT    offSegment;
    UINT    cbSegment;
    UINT    flags;
    UINT    cbAlloc;
    HGLOBAL h;
    UINT    alignShift;
    UINT    reserved[2];
} SEGINFO;
```

The **SEGINFO** structure contains information about a data or code segment. This structure is filled in by the **GetCodeInfo** function.

Members

offSegment
Specifies the offset, in sectors, to the contents of the segment data, relative to the beginning of the file. (Zero means no file data is available.) The size of the sector is determined by shifting left by 1 the value given in the **alignShift** member.

cbSegment
Specifies the length of the segment in the file, in bytes. Zero means 64K.

flags
Contains flags which specify attributes of the segment. The following list describes these flags:

Bit	Meaning
0–2	Specifies the segment type. If bit 0 is set to 1, the segment is a data segment. Otherwise, the segment is a code segment.
3	Specifies whether segment data is iterated. When this bit is set to 1, the segment data is iterated.
4	Specifies whether the segment is movable or fixed. When this bit is set to 1, the segment is movable. Otherwise, it is fixed.
5–6	Reserved.
7	Specifies whether the segment is a read-only data segment or an execute-only code segment. If this bit is set to 1 and the segment is a code segment, the segment is an execute-only segment. If this bit is set to zero and the segment is a data segment, it is a read-only segment.
8	Specifies whether the segment has associated relocation information. If this bit is set to 1, the segment has relocation information. Otherwise, the segment does not have relocation information.
9	Specifies whether the segment has debugging information. If this bit is set to 1, the segment has debugging information. Otherwise, the segment does not have debugging information.
10–15	Reserved.

cbAlloc
Specifies the total amount of memory allocated for the segment. This amount may exceed the actual size of the segment. Zero means 64K.

h
Identifies the global memory for the segment.

alignShift
Specifies the size of the addressable sector as an exponent of 2. An executable file pads the application's code, data, and resource segments with zero bytes so that the segments are always a multiple of the file-segment size. Windows discards the extra bytes when it loads the segments from the file.

reserved
Specifies two reserved **UINT** values.

See Also **GetCodeInfo**

SIZE

3.1

```
typedef struct tagSIZE {
    int cx;
    int cy;
} SIZE;
```

The **SIZE** structure contains viewport extents, window extents, text extents, bitmap dimensions, and the aspect-ratio filter for some extended functions for Windows 3.1.

Members

cx
Specifies the x-extent when a function returns.

cy
Specifies the y-extent when a function returns.

See Also

GetAspectRatioFilterEx, GetBitmapDimensionEx, GetTextExtentPoint, GetViewportExtEx, GetWindowExtEx, ScaleViewportExtEx, ScaleWindowExtEx, SetBitmapDimensionEx, SetViewportExtEx, SetWindowExtEx

STACKTRACEENTRY

3.1

```
#include <toolhelp.h>

typedef struct tagSTACKTRACEENTRY {  /* ste */
    DWORD   dwSize;
    HTASK   hTask;
    WORD    wSS;
    WORD    wBP;
    WORD    wCS;
    WORD    wIP;
    HMODULE hModule;
    WORD    wSegment;
    WORD    wFlags;
} STACKTRACEENTRY;
```

The **STACKTRACEENTRY** structure contains information about one stack frame. This information enables an application to trace back through the stack of a specific task.

Members

dwSize
Specifies the size of the **STACKTRACEENTRY** structure, in bytes.

hTask
Identifies the task handle for the stack.

wSS
Contains the value in the SS register. This value is used with the value of the **wBP** member to determine the next entry in the stack-trace table.

wBP
Contains the value in the BP register. This value is used with the **wSS** value to determine the next entry in the stack-trace table.

wCS
Contains the value in the CS register on return. This value is used with the value of the **wIP** member to determine the return value of the function.

wIP
Contains the value in the IP register on return. This value is used with the **wCS** value to determine the return value of the function.

hModule
Identifies the module that contains the currently executing function.

wSegment
Contains the segment number of the current selector.

wFlags
Indicates the frame type. This type can be one of the following values:

Value	Meaning
FRAME_FAR	The CS register contains a valid code segment.
FRAME_NEAR	The CS register is null.

See Also

StackTraceCSIPFirst, StackTraceNext, StackTraceFirst

SYSHEAPINFO

3.1

```
#include <toolhelp.h>

typedef struct tagSYSHEAPINFO {  /* shi */
    DWORD   dwSize;
    WORD    wUserFreePercent;
    WORD    wGDIFreePercent;
    HGLOBAL hUserSegment;
    HGLOBAL hGDISegment;
} SYSHEAPINFO;
```

The **SYSHEAPINFO** structure contains information about the USER and GDI modules.

Members

dwSize
Specifies the size of the **SYSHEAPINFO** structure, in bytes.

wUserFreePercent
Specifies the percentage of the USER local heap that is free.

wGDIFreePercent
Specifies the percentage of the GDI local heap that is free.

hUserSegment
Identifies the DGROUP segment of the USER local heap.

hGDISegment
Identifies the DGROUP segment of the GDI local heap.

See Also

SystemHeapInfo

TASKENTRY

3.1

```
#include <toolhelp.h>

typedef struct tagTASKENTRY {  /* te */
    DWORD     dwSize;
    HTASK     hTask;
    HTASK     hTaskParent;
    HINSTANCE hInst;
    HMODULE   hModule;
    WORD      wSS;
    WORD      wSP;
    WORD      wStackTop;
    WORD      wStackMinimum;
    WORD      wStackBottom;
    WORD      wcEvents;
    HGLOBAL   hQueue;
    char      szModule[MAX_MODULE_NAME + 1];
    WORD      wPSPOffset;
    HANDLE    hNext;
} TASKENTRY;
```

The **TASKENTRY** structure contains information about one task.

Members

dwSize
Specifies the size of the **TASKENTRY** structure, in bytes.

hTask
Identifies the task handle for the stack.

hTaskParent
Identifies the parent of the task.

hInst
Identifies the instance handle of the task. This value is equivalent to the task's DGROUP segment selector.

hModule
Identifies the module that contains the currently executing function.

wSS
Contains the value in the SS register.

wSP
Contains the value in the SP register.

wStackTop
Specifies the offset to the top of the stack (lowest address on the stack).

wStackMinimum
Specifies the lowest segment number of the stack during execution of the task.

wStackBottom
Specifies the offset to the bottom of the stack (highest address on the stack).

wcEvents
Specifies the number of pending events.

hQueue
Identifies the task queue.

szModule
Specifies the name of the module that contains the currently executing function.

wPSPOffset
Specifies the offset from the program segment prefix (PSP) to the beginning of the executable code segment.

hNext
Identifies the next entry in the task list. This member is reserved for internal use by Windows.

See Also **TaskFindHandle**, **TaskFirst**, **TaskNext**

TEXTMETRIC

2.x

```
typedef struct tagTEXTMETRIC {  /* tm */
    int  tmHeight;
    int  tmAscent;
    int  tmDescent;
    int  tmInternalLeading;
    int  tmExternalLeading;
    int  tmAveCharWidth;
    int  tmMaxCharWidth;
    int  tmWeight;
    BYTE tmItalic;
    BYTE tmUnderlined;
    BYTE tmStruckOut;
    BYTE tmFirstChar;
    BYTE tmLastChar;
    BYTE tmDefaultChar;
    BYTE tmBreakChar;
    BYTE tmPitchAndFamily;
    BYTE tmCharSet;
    int  tmOverhang;
    int  tmDigitizedAspectX;
    int  tmDigitizedAspectY;
} TEXTMETRIC;
```

The **TEXTMETRIC** structure contains basic information about a physical font. For Windows version 3.1, the **EnumFonts** and **EnumFontFamilies** functions return information about TrueType fonts in a **NEWTEXTMETRIC** structure.

Members

tmHeight
Specifies the height of character cells. (The height is the sum of the **tmAscent** and **tmDescent** members.)

tmAscent
Specifies the ascent of character cells. (The ascent is the space between the base line and the top of the character cell.)

tmDescent
Specifies the descent of character cells. (The descent is the space between the bottom of the character cell and the base line.)

tmInternalLeading
Specifies the difference between the point size of a font and the physical size of the font. For TrueType fonts, this value is equal to **tmHeight** minus (*s* * **ntmSizeEM**), where *s* is the scaling factor for the TrueType font and **ntmSizeEM** is a value from the **NEWTEXTMETRIC** structure. For bitmap fonts, this value is used to determine the point size of a font. When an application specifies a negative value in the **lfHeight** member of the **LOGFONT** structure, the application is requesting a font whose height equals **tmHeight** minus **tmInternalLeading**.

tmExternalLeading
Specifies the amount of extra leading (space) that the application adds between rows. Since this area is outside the character cell, it contains no marks and will not be altered by text output calls in either opaque or transparent mode. The font designer sometimes sets this member to zero.

tmAveCharWidth
Specifies the average width of characters in the font. For ANSI_CHARSET fonts, this is a weighted average of the characters "a" through "z" and the space character. For other character sets, this value is an unweighted average of all characters in the font.

tmMaxCharWidth
Specifies the "B" spacing of the widest character in the font. For more information about "B" spacing, see the description of the **ABC** structure.

tmWeight
Specifies the weight of the font. This member can be one of the following values:

Constant	Value
FW_DONTCARE	0
FW_THIN	100
FW_EXTRALIGHT	200

Constant	Value
FW_ULTRALIGHT	200
FW_LIGHT	300
FW_NORMAL	400
FW_REGULAR	400
FW_MEDIUM	500
FW_SEMIBOLD	600
FW_DEMIBOLD	600
FW_BOLD	700
FW_EXTRABOLD	800
FW_ULTRABOLD	800
FW_BLACK	900
FW_HEAVY	900

tmItalic
Specifies an italic font if it is nonzero.

tmUnderlined
Specifies an underlined font if it is nonzero.

tmStruckOut
Specifies a "struckout" font if it is nonzero.

tmFirstChar
Specifies the value of the first character defined in the font.

tmLastChar
Specifies the value of the last character defined in the font.

tmDefaultChar
Specifies the value of the character that will be substituted for characters that are not in the font.

tmBreakChar
Specifies the value of the character that will be used to define word breaks for text justification.

tmPitchAndFamily
Specifies the pitch and family of the selected font.

The four low-order bits identify the type of font, as shown in the following list:

Value	Meaning
TMPF_PITCH	Designates a fixed-pitch font.
TMPF_VECTOR	Designates a vector or TrueType font.
TMPF_TRUETYPE	Designates a TrueType font.
TMPF_DEVICE	Designates a device font.

Some fonts are identified by several of these bits—for example, the bits TMPF_PITCH, TMPF_VECTOR, and TMPF_TRUETYPE would be set for the monospace TrueType font, Courier New. The TMPF_DEVICE bit could be set for a TrueType font as well, because this bit is set for both downloaded and device-resident fonts.

When the TMPF_TRUETYPE bit is set, the font is usable on all output devices. For example, if a TrueType font existed on a printer but could not be used on the display, the TMPF_TRUETYPE bit would not be set for that font.

The four high-order bits of this member designate the font family. The **tmPitchAndFamily** member can be combined with the value 0xF0 by using the bitwise AND operator and can then be compared with the font family names for an identical match. The following font families are defined:

Value	Meaning
FF_DECORATIVE	Novelty fonts. Old English is an example.
FF_DONTCARE	Don't care or don't know.
FF_MODERN	Fonts with constant stroke width, with or without serifs. Pica, Elite, and Courier New are examples.
FF_ROMAN	Fonts with variable stroke width and with serifs. Times New Roman and New Century Schoolbook are examples.
FF_SCRIPT	Fonts designed to look like handwriting. Script and Cursive are examples.
FF_SWISS	Fonts with variable stroke width and without serifs. MS Sans Serif is an example.

tmCharSet

Specifies the character set of the font. The following values are defined:

Constant	Value
ANSI_CHARSET	0
DEFAULT_CHARSET	1
SYMBOL_CHARSET	2
SHIFTJIS_CHARSET	128
OEM_CHARSET	255

tmOverhang

Specifies the extra width that is added to some synthesized fonts. When synthesizing some attributes, such as bold or italic, GDI or a device sometimes adds width to a string on both a per-character and per-string basis. For example, GDI makes a string bold by expanding the intracharacter spacing and overstriking by an offset value and italicizes a font by skewing the string. In either case, the string is wider after the attribute is synthesized. For bold strings, the overhang is the distance by which the overstrike is offset. For italic strings, the overhang is the amount the top of the font is skewed past the bottom of the font.

The **tmOverhang** member is zero for many italic and bold TrueType fonts because many TrueType fonts include italic and bold faces that are not synthesized. For example, the overhang for Courier New Italic is zero.

An application that uses raster fonts can use the overhang value to determine the spacing between words that have different attributes.

tmDigitizedAspectX
Specifies the horizontal aspect of the device for which the font was designed.

tmDigitizedAspectY
Specifies the vertical aspect of the device for which the font was designed. The ratio of the **tmDigitizedAspectX** and **tmDigitizedAspectY** members is the aspect ratio of the device for which the font was designed.

Comments

All sizes are given in logical units; that is, they depend on the current mapping mode of the display context.

See Also

EnumFontFamilies, EnumFonts, GetDeviceCaps, GetTextMetrics

TIMERINFO

3.1

```
#include <toolhelp.h>

typedef struct tagTIMERINFO {  /* ti */
    DWORD dwSize;
    DWORD dwmsSinceStart;
    DWORD dwmsThisVM;
} TIMERINFO;
```

The **TIMERINFO** structure contains the elapsed time since the current task became active and since the virtual machine (VM) started.

Members

dwSize
Specifies the size of the **TIMERINFO** structure, in bytes.

dwmsSinceStart
Contains the amount of time, in milliseconds, since the current task became active.

dwmsThisVM
Contains the amount of time, in milliseconds, since the current VM started.

Comments

In standard mode, the **dwmsSinceStart** and **dwmsThisVM** values are the same.

See Also

TimerCount

TTPOLYCURVE

3.1

```
typedef struct tagTTPOLYCURVE {
    UINT    wType;
    UINT    cpfx;
    POINTFX apfx[1];
} TTPOLYCURVE;
```

The **TTPOLYCURVE** structure contains information about a curve in the outline of a TrueType character.

Members

wType
Specifies the type of curve described by the structure. This member can be one of the following values:

Value	Meaning
TT_PRIM_LINE	Curve is a polyline.
TT_PRIM_QSPLINE	Curve is a quadratic spline.

cpfx
Specifies the number of **POINTFX** structures in the array.

apfx
Specifies an array of **POINTFX** structures that define the polyline or quadratic spline.

Comments

When an application calls the **GetGlyphOutline** function, a glyph outline for a TrueType character is returned in a **TTPOLYGONHEADER** structure followed by as many **TTPOLYCURVE** structures as are required to describe the glyph. All points are returned as **POINTFX** structures and represent absolute positions, not relative moves. The starting point given by the **pfxStart** member of the **TTPOLYGONHEADER** structure is the point at which the outline for a contour begins. The **TTPOLYCURVE** structures that follow can be either polyline records or spline records.

Polyline records are a series of points; lines drawn between the points describe the outline of the character. Spline records represent the quadratic curves used by TrueType (that is, quadratic b-splines).

See Also

POINTFX, TTPOLYGONHEADER

TTPOLYGONHEADER

3.1

```
typedef struct tagTTPOLYGONHEADER {
    DWORD   cb;
    DWORD   dwType;
    POINTFX pfxStart;
} TTPOLYGONHEADER;
```

The **TTPOLYGONHEADER** structure specifies the starting position and type of a contour in a TrueType character outline.

Members

cb
Specifies the number of bytes required by the **TTPOLYGONHEADER** structure and **TTPOLYCURVE** structure or structures required to describe the contour.

dwType
Specifies the type of character outline that is returned. Currently, this value must be TT_POLYGON_TYPE.

pfxStart
Specifies the starting point of the contour in the character outline.

Comments

Each **TTPOLYGONHEADER** structure is followed by one or more **TTPOLYCURVE** structures.

See Also

POINTFX, **TTPOLYCURVE**

VS_FIXEDFILEINFO

3.1

```
#include <ver.h>

typedef struct tagVS_FIXEDFILEINFO {    /* vsffi */
    DWORD dwSignature;
    DWORD dwStrucVersion;
    DWORD dwFileVersionMS;
    DWORD dwFileVersionLS;
    DWORD dwProductVersionMS;
    DWORD dwProductVersionLS;
    DWORD dwFileFlagsMask;
    DWORD dwFileFlags;
    DWORD dwFileOS;
    DWORD dwFileType;
    DWORD dwFileSubtype;
    DWORD dwFileDateMS;
    DWORD dwFileDateLS;
} VS_FIXEDFILEINFO;
```

The **VS_FIXEDFILEINFO** structure contains version information about a file.

Members

dwSignature
Specifies the value 0xFEEFO4BD.

dwStrucVersion
Specifies the binary version number of this structure. The high-order word contains the major version number, and the low-order word contains the minor version number. This value must be greater than 0x00000029.

dwFileVersionMS
Specifies the high-order 32 bits of the binary version number for the file. The value of this member is used with the value of the **dwFileVersionLS** member to form a 64-bit version number.

dwFileVersionLS
Specifies the low-order 32 bits of the binary version number for the file. The value of this member is used with the **dwFileVersionMS** value to form a 64-bit version number.

dwProductVersionMS
Specifies the high-order 32 bits of the binary version number of the product with which the file is distributed. The value of this member is used with the value of the **dwProductVersionLS** member to form a 64-bit version number.

dwProductVersionLS
Specifies the low-order 32 bits of the binary version number of the product with which the file is distributed. The value of this member is used with the **dwProductVersionMS** value to form a 64-bit version number.

dwFileFlagsMask

Specifies which bits in the **dwFileFlags** member are valid. If a bit is set, the corresponding bit in the **dwFileFlags** member is valid.

dwFileFlags

Specifies the Boolean attributes of the file. The attributes can be a combination of the following values:

Value	Meaning
VS_FF_DEBUG	File contains debugging information or is compiled with debugging features enabled.
VS_FF_INFOINFERRED	File contains a dynamically created version-information resource. Some of the blocks for the resource may be empty or incorrect. This value is not intended to be used in version-information resources created by using the **VERSIONINFO** statement.
VS_FF_PATCHED	File has been modified and is not identical to the original shipping file of the same version number.
VS_FF_PRERELEASE	File is a development version, not a commercially released product.
VS_FF_PRIVATEBUILD	File was not built using standard release procedures. If this value is given, the **StringFileInfo** block must contain a **PrivateBuild** string.
VS_FF_SPECIALBUILD	File was built by the original company using standard release procedures but is a variation of the standard file of the same version number. If this value is given, the **StringFileInfo** block must contain a **SpecialBuild** string.

dwFileOS

Specifies the operating system for which this file was designed. This member can be one of the following values:

Value	Meaning
VOS_UNKNOWN	Operating system for which the file was designed is unknown to Windows.
VOS_DOS	File was designed for MS-DOS.
VOS_NT	File was designed for Windows NT.
VOS_WINDOWS16	File was designed for Windows version 3.0 or later.
VOS_WINDOWS32	File was designed for 32-bit Windows.
VOS_DOS_WINDOWS16	File was designed for Windows version 3.0 or later running with MS-DOS.
VOS_DOS_WINDOWS32	File was designed for 32-bit Windows running with MS-DOS.
VOS_NT_WINDOWS32	File was designed for 32-bit Windows running with Windows NT.

The values 0x00002L, 0x00003L, 0x20000L and 0x30000L are reserved.

dwFileType

Specifies the general type of file. This type can be one of the following values:

Value	Meaning
VFT_UNKNOWN	File type is unknown to Windows.
VFT_APP	File contains an application.
VFT_DLL	File contains a dynamic-link library (DLL).
VFT_DRV	File contains a device driver. If the **dwFileType** member is VFT_DRV, the **dwFileSubtype** member contains a more specific description of the driver.
VFT_FONT	File contains a font. If the **dwFileType** member is VFT_FONT, the **dwFileSubtype** member contains a more specific description of the font.
VFT_VXD	File contains a virtual device.
VFT_STATIC_LIB	File contains a static-link library.

All other values are reserved for use by Microsoft.

dwFileSubtype

Specifies the function of the file. This member is zero unless the **dwFileType** member is VFT_DRV, VFT_FONT, or VFT_VXD.

If **dwFileType** is VFT_DRV, **dwFileSubtype** may be one of the following values:

Value	Meaning
VFT2_UNKNOWN	Driver type is unknown to Windows.
VFT2_DRV_COMM	File contains a communications driver.
VFT2_DRV_PRINTER	File contains a printer driver.
VFT2_DRV_KEYBOARD	File contains a keyboard driver.
VFT2_DRV_LANGUAGE	File contains a language driver.
VFT2_DRV_DISPLAY	File contains a display driver.
VFT2_DRV_MOUSE	File contains a mouse driver.
VFT2_DRV_NETWORK	File contains a network driver.
VFT2_DRV_SYSTEM	File contains a system driver.
VFT2_DRV_INSTALLABLE	File contains an installable driver.
VFT2_DRV_SOUND	File contains a sound driver.

If **dwFileType** is VFT_FONT, **dwFileSubtype** may be one of the following values:

Value	Meaning
VFT2_UNKNOWN	Font type is unknown to Windows.
VFT2_FONT_RASTER	File contains a raster font.
VFT2_FONT_VECTOR	File contains a vector font.
VFT2_FONT_TRUETYPE	File contains a TrueType font.

If **dwFileType** is VFT_VXD, **dwFileSubtype** contains the virtual-device identifier included in the virtual-device control block.

All **dwFileSubtype** values not listed here are reserved for use by Microsoft.

dwFileDateMS
Specifies the high-order 32 bits of a binary date/time stamp for the file. The value of this member is used with the value of the **dwFileDateLS** member to form a 64-bit number representing the date and time the file was created.

dwFileDateLS
Specifies the low-order 32 bits of a binary date/time stamp for the file. The value of this member is used with the **dwFileDateMS** value to form a 64-bit number representing the date and time the file was created.

Comments

The binary version numbers specified in this structure are intended to be integers rather than character strings. For a file or product that has decimal points or letters in its version number, the corresponding binary version number should be a reasonable numeric representation.

A third-party developer can use the file-version values to reflect a private version-numbering scheme, as long as each new version of the product has a higher number than the previous version. The File Installation library functions use these values when comparing the ages of files.

Microsoft Windows Resource Compiler sets the **dwFileDateMS** and **dwFileDateLS** members to zero.

See Also

VerQueryValue

WINDEBUGINFO

3.1

```
typedef struct tagWINDEBUGINFO {
    UINT    flags;
    DWORD   dwOptions;
    DWORD   dwFilter;
    char    achAllocModule[8];
    DWORD   dwAllocBreak;
    DWORD   dwAllocCount;
} WINDEBUGINFO;
```

The **WINDEBUGINFO** structure contains current system-debugging information for the debugging version of Windows 3.1.

Members

flags

Specifies which members of the **WINDEBUGINFO** structure are valid. This member can be one or more of the following values:

Value	Meaning
WDI_OPTIONS	**dwOptions** member is valid.
WDI_FILTER	**dwFilter** member is valid.
WDI_ALLOCBREAK	**achAllocModule**, **dwAllocBreak**, and **dwAllocCount** members are valid.

dwOptions

Specifies debugging options. This member is valid only if WDI_OPTIONS is specified in the **flags** member. It can be one or more of the following values:

Constant	Value	Meaning
DBO_CHECKHEAP	0x0001	Performs local heap checking after all calls to functions that manipulate local memory.
DBO_BUFFERFILL	0x0004	Fills buffers passed to API functions with 0xF9. This ensures that the supplied buffer is completely writable and helps detect overwrite problems when the supplied buffer size is not large enough.
DBO_DISABLEGPTRAPPING	0x0010	Disables hooking of the fault interrupt vectors. This option is not typically used by application developers, because parameter validation can cause many spurious traps that are not errors.

Constant	Value	Meaning
DBO_CHECKFREE	0x0020	Fills all freed local memory with 0xFB. All newly allocated memory is checked to ensure that it is still filled with 0xFB—this ensures that no application has written into a freed memory object. This option has no effect if DBO_CHECKHEAP is not specified.
DBO_INT3BREAK	0x0100	Breaks to the debugger with simple INT 3 rather than a call to the **FatalExit** function. This option does not generate a stack backtrace.
DBO_NOFATALBREAK	0x0400	Does not break with the "abort, break, ignore" prompt if a DBF_FATAL message occurs.
DBO_NOERRORBREAK	0x0800	Does not break with the "abort, break, ignore" prompt if a DBF_ERROR message occurs. This option also applies to invalid parameter errors.
DBO_WARNINGBREAK	0x1000	Breaks with the "abort, break, ignore" prompt if a DBF_WARNING message occurs. (Normally, DBF_WARNING messages are displayed but no break occurs). This option also applies to invalid parameter warnings.
DBO_TRACEBREAK	0x2000	Breaks with the "abort, break, ignore" on any DBF_TRACE message that matches the value specified in the **dwFilter** member.
DBO_SILENT	0x8000	Does not display warning, error, or fatal messages except in cases where a stack trace and "abort, break, ignore" prompt would occur.

dwFilter

Specifies filtering options for DBF_TRACE messages. (Normally, trace messages are not sent to the debug terminal.) This member can be one or more of the following values:

Constant	Value	Meaning
DBF_KRN_MEMMAN	0x0001	Enables KERNEL messages related to local and global memory management.
DBF_KRN_LOADMODULE	0x0002	Enables KERNEL messages related to module loading.
DBF_KRN_SEGMENTLOAD	0x0004	Enables KERNEL messages related to segment loading.
DBF_APPLICATION	0x0008	Enables trace messages originating from an application.
DBF_DRIVER	0x0010	Enables trace messages originating from device drivers.
DBF_PENWIN	0x0020	Enables trace messages originating from PENWIN.
DBF_MMSYSTEM	0x0040	Enables trace messages originating from MMSYSTEM.
DBF_GDI	0x0400	Enables trace messages originating from GDI.
DBF_USER	0x0800	Enables trace messages originating from USER.
DBF_KERNEL	0x1000	Enables any trace message originating from KERNEL. (This is a combination of DBF_KRN_MEMMAN, DBF_KRN_LOADMODULE, and DBF_KRN_SEGMENTLOAD.)

achAllocModule
Specifies the name of the application module. (This can be different from the name of the executable file.) This cannot be the name of a dynamic-link library (DLL). The name is limited to 8 characters.

dwAllocBreak
Specifies the number of global or local memory allocations to allow before failing allocation requests. When the count of allocations reaches the number specified in this member, that allocation and all subsequent allocations fail. If this member is zero, no allocation break is set, but the system counts allocations and reports the current count in the **dwAllocCount** member.

dwAllocCount
Current count of allocations. (This information is typically retrieved by calling the **GetWinDebugInfo** function.)

Comments

Developers can use the **achAllocModule**, **dwAllocBreak**, and **dwAllocCount** members to ensure that an application performs correctly in out-of-memory conditions. Because memory allocations made by the system fail once the break count is

reached, calls to functions such as **CreateWindow**, **CreateBrush**, and **SelectObject** will fail as well. Only allocations made within the context of the application specified by the **achAllocModule** member are affected by the allocation break count.

See Also **DebugOutput**, **GetWinDebugInfo**, **SetWinDebugInfo**

WINDOWPLACEMENT

3.1

```
typedef struct tagWINDOWPLACEMENT {     /* wndpl */
    UINT  length;
    UINT  flags;
    UINT  showCmd;
    POINT ptMinPosition;
    POINT ptMaxPosition;
    RECT  rcNormalPosition;
} WINDOWPLACEMENT;
```

The **WINDOWPLACEMENT** structure contains information about the placement of a window on the screen.

Members

length

Specifies the length, in bytes, of the structure. (The **GetWindowPlacement** function returns an error if this member is not specified correctly.)

flags

Specifies flags that control the position of the minimized window and the method by which the window is restored. This member can be one or both of the following flags:

Value	Meaning
WPF_SETMINPOSITION	Specifies that the x- and y-positions of the minimized window may be specified. This flag must be specified if the coordinates are set in the **ptMinPosition** member.
WPF_RESTORETOMAXIMIZED	Specifies that the restored window will be maximized, regardless of whether it was maximized before it was minimized. This setting is valid only the next time the window is restored. It does not change the default restoration behavior. This flag is valid only when the SW_SHOWMINIMIZED value is specified for the **showCmd** member.

showCmd
Specifies the current show state of the window. This member may be one of the following values:

Value	Meaning
SW_HIDE	Hides the window and passes activation to another window.
SW_MINIMIZE	Minimizes the specified window and activates the top-level window in the system's list.
SW_RESTORE	Activates and displays a window. If the window is minimized or maximized, Windows restores it to its original size and position (same as SW_SHOWNORMAL).
SW_SHOW	Activates a window and displays it in its current size and position.
SW_SHOWMAXIMIZED	Activates a window and displays it as a maximized window.
SW_SHOWMINIMIZED	Activates a window and displays it as an icon.
SW_SHOWMINNOACTIVE	Displays a window as an icon. The window that is currently active remains active.
SW_SHOWNA	Displays a window in its current state. The window that is currently active remains active.
SW_SHOWNOACTIVATE	Displays a window in its most recent size and position. The window that is currently active remains active.
SW_SHOWNORMAL	Activates and displays a window. If the window is minimized or maximized, Windows restores it to its original size and position (same as SW_RESTORE).

ptMinPosition
Specifies the position of the window's top-left corner when the window is minimized.

ptMaxPosition
Specifies the position of the window's top-left corner when the window is maximized.

rcNormalPosition
Specifies the window's coordinates when the window is in the normal (restored) position.

See Also **POINT, RECT, ShowWindow**

WINDOWPOS

3.1

```
typedef struct tagWINDOWPOS { /* wp */
    HWND    hwnd;
    HWND    hwndInsertAfter;
    int     x;
    int     y;
    int     cx;
    int     cy;
    UINT    flags;
} WINDOWPOS;
```

The **WINDOWPOS** structure contains information about the size and position of a window.

Members

hwnd
Identifies the window.

hwndInsertAfter
Identifies the window behind which this window is placed.

x
Specifies the position of the left edge of the window.

y
Specifies the position of the right edge of the window.

cx
Specifies the window width.

cy
Specifies the window height.

flags
Specifies window-positioning options. This member can be one of the following values:

Value	Meaning
SWP_DRAWFRAME	Draws a frame (defined in the class description for the window) around the window. The window receives a WM_NCCALCSIZE message.
SWP_HIDEWINDOW	Hides the window.
SWP_NOACTIVATE	Does not activate the window.
SWP_NOMOVE	Retains current position (ignores the **x** and **y** members).
SWP_NOOWNERZORDER	Does not change the owner window's position in the Z order.
SWP_NOSIZE	Retains current size (ignores the **cx** and **cy** members).

Value	Meaning
SWP_NOREDRAW	Does not redraw changes.
SWP_NOREPOSITION	Same as SWP_NOOWNERZORDER.
SWP_NOZORDER	Retains current ordering (ignores the **hwndInsertAfter** member).
SWP_SHOWWINDOW	Displays the window.

See Also **EndDeferWindowPos**

WNDCLASS

2.x

```
typedef struct tagWNDCLASS {    /* wc */
    UINT      style;
    WNDPROC   lpfnWndProc;
    int       cbClsExtra;
    int       cbWndExtra;
    HINSTANCE hInstance;
    HICON     hIcon;
    HCURSOR   hCursor;
    HBRUSH    hbrBackground;
    LPCSTR    lpszMenuName;
    LPCSTR    lpszClassName;
} WNDCLASS;
```

The **WNDCLASS** structure contains the class attributes that are registered by the **RegisterClass** function.

Members

style

Specifies the class style. These styles can be combined by using the bitwise OR operator. This can be any combination of the following values:

Value	Meaning
CS_BYTEALIGNCLIENT	Aligns the client area of a window on the byte boundary (in the x-direction).
CS_BYTEALIGNWINDOW	Aligns a window on the byte boundary (in the x-direction). This flag should be set by applications that perform bitmap operations in windows by using the **BitBlt** function.
CS_CLASSDC	Gives the window class its own display context (shared by instances).
CS_DBLCLKS	Sends double-click messages to a window.

Value	Meaning
CS_GLOBALCLASS	Specifies that the window class is an application global class. An application global class is created by an application or library and is available to all applications. The class is destroyed when the application or library that created the class exits; it is essential, therefore, that all windows created with the application global class be closed before this occurs.
CS_HREDRAW	Redraws the entire window if the horizontal size changes.
CS_NOCLOSE	Inhibits the close option on the System menu.
CS_OWNDC	Gives each window instance its own display context. Note that although the CS_OWNDC style is convenient, it must be used with discretion because each display context occupies approximately 800 bytes of memory.
CS_PARENTDC	Gives the display context of the parent window to the window class.
CS_SAVEBITS	Specifies that the system should try to save the screen image behind a window created from this window class as a bitmap. Later, when the window is removed, the system uses the bitmap to quickly restore the screen image. This style is useful for small windows that are displayed briefly and then removed before much other screen activity takes place (for example, menus or dialog boxes). This style increases the time required to display the window since the system must first allocate memory to store the bitmap.
CS_VREDRAW	Redraws the entire window if the vertical size changes.

lpfnWndProc
Points to the window procedure. For more information, see the description of the **WindowProc** callback function.

cbClsExtra
Specifies the number of bytes to allocate following the window-class structure. These bytes are initialized to zero.

cbWndExtra
Specifies the number of bytes to allocate following the window instance. These bytes are initialized to zero. If an application uses the **WNDCLASS** structure to register a dialog box created with the **CLASS** directive in the resource file, it must set this member to DLGWINDOWEXTRA.

hInstance
Identifies the class module. This member must be an instance handle and must not be NULL.

hIcon
Identifies the class icon. This member must be a handle to an icon resource. If this member is NULL, the application must draw an icon whenever the user minimizes the application's window.

hCursor
Identifies the class cursor. This member must be a handle to a cursor resource. If this member is NULL, the application must explicitly set the cursor shape whenever the mouse moves into the application's window.

hbrBackground
Identifies the class background brush. This member can be either a handle to the physical brush that is to be used for painting the background, or it can be a color value. If a color value is given, it must be one of the standard system colors listed below, and the value 1 must be added to the chosen color (for example, COLOR_BACKGROUND + 1 specifies the system background color). If a color value is given, it must be converted to one of the following **HBRUSH** types:

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

The system automatically deletes class background brushes when the class is freed. An application should not delete these brushes, because a class may be used by multiple instances of the application.

When this member is NULL, the application must paint its own background whenever it is requested to paint in its client area. The application can determine when the background needs painting by processing the message WM_ERASEBKGND or by testing the **fErase** member of the **PAINTSTRUCT** structure filled by the **BeginPaint** function.

lpszMenuName
Points to a null-terminated string that specifies the resource name of the class menu (as the name appears in the resource file). If an integer is used to identify the menu, the **MAKEINTRESOURCE** macro can be used. If this member is NULL, windows belonging to this class have no default menu.

lpszClassName
Points to a null-terminated string that specifies the name of the window class.

See Also PAINTSTRUCT

Macros

Chapter 4

This chapter describes the purpose and defines the parameters of macros associated with functions and structures for the Microsoft Windows operating system, version 3.1. It lists the Windows macros in alphabetic order.

DECLARE_HANDLE

3.1

DECLARE_HANDLE(*name*)

The **DECLARE_HANDLE** macro creates a data type that can be used to define 16-bit handles.

Parameters

name
Specifies the name of the new data type.

Comments

The **DECLARE_HANDLE** macro is defined in WINDOWS.H as follows:

```
#define DECLARE_HANDLE(name) struct name##__ { int unused; }; \
                             typedef const struct name##__ NEAR* name
```

See Also

DECLARE_HANDLE32

DECLARE_HANDLE32

3.1

#include <ddeml.h>

DECLARE_HANDLE32(*name*)

The **DECLARE_HANDLE32** macro creates a data type that can be used to define 32-bit handles.

Parameters

name
Specifies the name of the new data type.

Comments The **DECLARE_HANDLE32** macro is defined in DDEML.H as follows:

```
#define DECLARE_HANDLE32(name) struct name##__ { int unused; }; \
                               typedef const struct name##__ _far* name
```

See Also **DECLARE_HANDLE**

FIELDOFFSET

3.1

int FIELDOFFSET(*type*, *field*)

The **FIELDOFFSET** macro computes the address offset of the specified member in the structure specified by the *type* parameter.

Parameters

type
Specifies the name of the structure.

field
Specifies the name of the member defined within the given structure.

Return Value The return value is the address offset of the given structure member.

Comments The **FIELDOFFSET** macro is defined in WINDOWS.H as follows:

```
#define FIELDOFFSET(type, field)    ((int)(&((type NEAR*)1)->field)-1)
```

GetBValue

3.1

BYTE GetBValue(*rgb*)
DWORD *rgb*; /* RGB color value */

The **GetBValue** macro extracts the intensity value of the blue color field from the 32-bit integer value specified by the *rgb* parameter.

Parameters

rgb
Specifies the RGB color value.

Return Value

The return value specifies the intensity of the blue color field.

Comments

The **GetBValue** macro is defined in WINDOWS.H as follows:

```
#define GetBValue(rgb)  ((BYTE)((rgb)>>16))
```

See Also

GetGValue, **GetRValue**, **RGB**

GetGValue

3.1

BYTE GetGValue(*rgb*)
DWORD *rgb*; /* RGB color value */

The **GetGValue** macro extracts the intensity value of the green color field from the 32-bit integer value specified by the *rgb* parameter.

Parameters

rgb
Specifies the RGB color value.

Return Value

The return value specifies the intensity of the green color field.

Comments

The **GetGValue** macro is defined in WINDOWS.H as follows:

```
#define GetGValue(rgb)  ((BYTE)(((WORD)(rgb)) >> 8))
```

See Also

GetBValue, **GetRValue**, **RGB**

GetRValue

3.1

BYTE GetRValue(*rgb***)**
DWORD *rgb*; /* RGB color value */

The **GetRValue** macro extracts the intensity value of the red color field from the 32-bit integer value specified by the *rgb* parameter.

Parameters

rgb
Specifies the RGB color value.

Return Value

The return value specifies the intensity of the red color field.

Comments

The **GetRValue** macro is defined in WINDOWS.H as follows:

```
#define GetRValue(rgb)  ((BYTE)(rgb))
```

See Also

GetBValue, **GetGValue**, **RGB**

GlobalDiscard

2.x

HGLOBAL GlobalDiscard(*hglb***)**
HGLOBAL *hglb*; /* handle of object to discard */

The **GlobalDiscard** macro discards the given global memory object. The lock count of the memory object must be zero.

Parameters

hglb
Identifies the global memory object to be discarded.

Return Value

The return value is a handle of the discarded object if the macro is successful. Otherwise, it is NULL.

Comments

The **GlobalDiscard** macro discards only global objects that an application allocated with the GMEM_DISCARDABLE and GMEM_MOVEABLE flags set. The macro fails if an application attempts to discard a fixed or locked object.

Although **GlobalDiscard** removes the global memory object from memory, the object's handle remains valid. An application can subsequently pass the handle to the **GlobalReAlloc** function to allocate another global memory object identified by the same handle.

The **GlobalDiscard** macro is defined in WINDOWS.H as follows:

```
#define GlobalDiscard(h)  GlobalReAlloc(h, 0L, GMEM_MOVEABLE)
```

See Also **GlobalReAlloc**

HIBYTE

2.x

BYTE HIBYTE(*wInteger***)**
WORD *wInteger*; /* value from which high byte is retrieved */

The **HIBYTE** macro retrieves the high-order byte from the integer value specified by the *wInteger* parameter.

Parameters

wInteger
Specifies the value to be converted.

Return Value

The return value specifies the high-order byte of the given value.

Comments

The **HIBYTE** macro is defined in WINDOWS.H as follows:

```
#define HIBYTE(w)  ((BYTE)(((WORD)(w) >> 8) & 0xFF))
```

HIWORD

2.x

WORD HIWORD(*dwInteger***)**
DWORD *dwInteger*; /* value from which high word is retrieved */

The **HIWORD** macro retrieves the high-order word from the 32-bit integer value specified by the *dwInteger* parameter.

Parameters

dwInteger
Specifies the value to be converted.

Return Value

The return value specifies the high-order word of the given 32-bit integer value.

Comments

The **HIWORD** macro is defined in WINDOWS.H as follows:

```
#define HIWORD(l)  ((WORD)((((DWORD)(l)) >> 16) & 0xFFFF))
```

LOBYTE

2.x

BYTE LOBYTE(*wVal*)
WORD *wVal*; /* value from which low byte is retrieved */

The **LOBYTE** macro extracts the low-order byte from the short-integer value specified by the *wVal* parameter.

Parameters

wVal
Specifies the value to be converted.

Return Value

The return value specifies the low-order byte of the value.

Comments

The **LOBYTE** macro is defined in WINDOWS.H as follows:

```
#define LOBYTE(w)  ((BYTE)(w))
```

See Also

LOWORD

LocalDiscard

2.x

HLOCAL LocalDiscard(*hloc*)
HLOCAL *hloc*; /* handle of object to discard */

The **LocalDiscard** macro discards the given local memory object. The lock count of the memory object must be zero.

Parameters

hloc
Identifies the local memory object to be discarded.

Return Value

The return value is equal to the *hloc* parameter if the macro is successful. Otherwise, it is NULL.

Comments

Although the **LocalDiscard** macro removes the local memory object from memory, the object's handle remains valid. An application can subsequently pass the handle to the **LocalReAlloc** function to allocate another local memory object identified by the same handle.

The **LocalLock** function increments (increases by one) a memory object's lock count. The **LocalUnlock** function decrements (decreases by one) the lock count.

The **LocalDiscard** macro is defined in WINDOWS.H as follows:

```
#define LocalDiscard(h)   LocalReAlloc(h, 0, LMEM_MOVEABLE)
```

See Also **LocalLock, LocalReAlloc, LocalUnlock**

LockData

2.x

HANDLE LockData(*dummy*)

The **LockData** macro locks the current data segment in memory. It is intended to be used in modules that have movable data segments.

Parameters

dummy
This parameter is ignored.

Return Value The return value identifies the locked data segment if the function is successful. Otherwise, it is NULL.

Comments The **LockData** macro is defined in WINDOWS.H as follows:

```
#define LockData(dummy)  LockSegment((UINT)-1)
```

See Also **LockSegment**

LOWORD

2.x

WORD LOWORD(*dwVal*)
DWORD *dwVal*; /* value from which low word is retrieved */

The **LOWORD** macro extracts the low-order word from the 32-bit integer value specified by the *dwVal* parameter.

Parameters

dwVal
Specifies the value to be converted.

Return Value The return value specifies the low-order word of the 32-bit integer value.

Comments The **LOWORD** macro is defined in WINDOWS.H as follows:

```
#define LOWORD(l)  ((WORD)(DWORD)(l))
```

See Also **LOBYTE**

MAKEINTATOM

2.x

LPCSTR MAKEINTATOM(*wInteger***)**
WORD *wInteger*; /* integer to make into atom */

The **MAKEINTATOM** macro creates an integer atom that represents a character string of decimal digits.

Integer atoms created by this macro can be added to the atom table using the **AddAtom** function.

Parameters

wInteger
Specifies the numeric value to be made into an integer atom.

Return Value The return value is a pointer to the atom created for the given integer.

Comments Although the return value of the **MAKEINTATOM** macro is cast as an **LPCSTR**, the return value cannot be used as a string pointer, except when it is passed to atom-management functions that require an **LPCSTR** parameter.

The **DeleteAtom** function always succeeds for integer atoms, even though it does nothing. The string returned by the **GetAtomName** function for an integer atom will be a null-terminated string where the first character is a pound sign (#) and the remaining characters are the word used in the **MAKEINTATOM** macro.

The **MAKEINTATOM** macro is defined in WINDOWS.H as follows:

```
#define MAKEINTATOM(i)  ((LPCSTR)MAKELP(NULL, (i)))
```

Example The following example uses the **MAKEINTATOM** macro to convert the number 32,565 into an integer atom. The atom is then added to the local atom table by the **AddAtom** function:

```
ATOM at;
char szMsg[80];
LPCSTR lpszAtom;

lpszAtom = MAKEINTATOM(32565);
at = AddAtom(lpszAtom);

if (at == 0)
    MessageBox(hwnd, "AddAtom failed", "", MB_ICONSTOP);
else {
    sprintf(szMsg, "AddAtom returned %u", at);
    MessageBox(hwnd, szMsg, "", MB_OK);
}
```

See Also **AddAtom, DeleteAtom, GetAtomName**

MAKEINTRESOURCE

2.x

LPCSTR MAKEINTRESOURCE(*idResource***)**
WORD *idResource*; /* resource identifier to convert */

The **MAKEINTRESOURCE** macro converts an integer resource identifier into a value compatible with Windows resource-management functions. This macro is used in place of a string containing the name of the resource.

Parameters

idResource
Specifies the integer resource identifier to be converted.

Return Value

The return value contains the *idResource* parameter in the low-order word and zero in the high-order word.

Comments

The **MAKEINTRESOURCE** macro is defined in WINDOWS.H as follows:

```
#define MAKEINTRESOURCE(i)  ((LPCSTR)MAKELP(NULL, (i)))
```

See Also **MAKELP**

MAKELONG

2.x

DWORD MAKELONG(*wLow*, *wHigh*)
WORD *wLow*; /* low-order word of long value */
WORD *wHigh*; /* high-order word of long value */

The **MAKELONG** macro creates an unsigned long integer by concatenating two integer values, specified by the *wLow* and *wHigh* parameters.

Parameters

wLow
Specifies the low-order word of the new long value.

wHigh
Specifies the high-order word of the new long value.

Return Value

The return value specifies an unsigned long-integer value.

Comments

The **MAKELONG** macro is defined in WINDOWS.H as follows:

```
#define MAKELONG(low, high) \
    ((LONG)(((WORD)(low)) | (((DWORD)((WORD)(high))) << 16)))
```

MAKELP

3.1

void FAR* MAKELP(*wSel*, *wOff*)
WORD *wSel*; /* selector */
WORD *wOff*; /* offset */

The **MAKELP** macro combines a segment selector and an address offset to create a long (32-bit) pointer to a memory address.

Parameters

wSel
Specifies a segment selector.

wOff
Specifies an offset from the beginning of the given segment to the desired byte.

Return Value

The return value is a long pointer to an unspecified data type.

Comments

The **MAKELP** macro is defined in WINDOWS.H as follows:

```
#define MAKELP(sel, off)    ((void FAR*)MAKELONG((off), (sel)))
```

See Also

MAKELONG

MAKELPARAM

3.1

LPARAM MAKELPARAM(*wLow*, *wHigh*)
WORD *wLow*; /* low-order word */
WORD *wHigh*; /* high-order word */

The **MAKELPARAM** macro creates an unsigned long integer for use as an *lParam* parameter in a message. The macro concatenates two integer values, specified by the *wLow* and *wHigh* parameters.

Parameters

wLow
Specifies the low-order word of the new long value.

wHigh
Specifies the high-order word of the new long value.

Return Value

The return value specifies an unsigned long-integer value.

Comments

The **MAKELPARAM** macro is defined in WINDOWS.H as follows:

```
#define MAKELPARAM(low, high)   ((LPARAM)MAKELONG(low, high))
```

See Also

MAKELONG, **MAKELRESULT**

MAKELRESULT

3.1

LRESULT MAKELRESULT(*wLow*, *wHigh*)
WORD *wLow*; /* low-order word */
WORD *wHigh*; /* high-order word */

The **MAKELRESULT** macro creates an unsigned long integer for use as a return value from a window procedure. The macro concatenates two integer values, specified by the *wLow* and *wHigh* parameters.

Parameters

wLow
Specifies the low-order word of the new long value.

wHigh
Specifies the high-order word of the new long value.

Return Value

The return value specifies an unsigned long-integer value.

Comments The **MAKELRESULT** macro is defined in WINDOWS.H as follows:

```
#define MAKELRESULT(low, high)  ((LRESULT)MAKELONG(low, high))
```

See Also **MAKELONG, MAKELPARAM**

MAKEPOINT

2.x

POINT MAKEPOINT(*lval*)
DWORD *lval*; /* coordinates of a point */

The **MAKEPOINT** macro converts a long value that contains the x- and y-coordinates of a point into a **POINT** structure. This macro is useful for converting the long value returned by the **GetMessagePos** function into a **POINT** structure and for converting the *lParam* value passed with mouse messages into a **POINT** structure containing the mouse coordinates.

Parameters

lval
Specifies the coordinates of a point. The x-coordinate is in the low-order word, and the y-coordinate is in the high-order word.

Return Value The return value is a pointer to a **POINT** structure.

Comments The **MAKEPOINT** macro is defined in WINDOWS.H as follows:

```
#define MAKEPOINT(l)  (*((POINT FAR*)&(l)))
```

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

```
typedef struct tagPOINT {   /* pt */
   int x;
   int y;
} POINT;
```

The **MAKEPOINT** macro is not compatible with the Windows 32-bit application programming interface (API).

See Also **GetMessagePos**

max

2.x

int max(*value1*, *value2*)

The **max** macro compares two values and returns the value of the larger one. The data type can be any numerical data type, signed or unsigned. The type of the arguments and the return value is the same.

Parameters

value1
Specifies the first of two values.

value2
Specifies the second of two values.

Return Value

The return value is *value1* or *value2*, whichever is greater.

Comments

The **max** macro is defined in WINDOWS.H as follows:

```
#define max(a, b)  (((a) > (b)) ? (a) : (b))
```

See Also

min

min

2.x

int min(*value1*, *value2*)

The **min** macro compares two values and returns the value of the smaller one. The data type can be any numerical data type, signed or unsigned. The type of the arguments and the return value is the same.

Parameters

value1
Specifies the first of two values.

value2
Specifies the second of two values.

Return Value

The return value is *value1* or *value2*, whichever is smaller.

Comments The **min** macro is defined in WINDOWS.H as follows:

```
#define min(a, b)  (((a) < (b)) ? (a) : (b))
```

See Also **max**

OFFSETOF

3.1

WORD OFFSETOF(*lp*)
void FAR* *lp*; /* long pointer */

The **OFFSETOF** macro retrieves the address offset of the specified long pointer.

Parameters

lp
Specifies a long pointer.

Return Value The return value is the offset address.

Comments The **OFFSETOF** macro is defined in WINDOWS.H as follows:

```
#define OFFSETOF(lp)        LOWORD(lp)
```

See Also **LOWORD, SELECTOROF**

PALETTEINDEX

3.0

COLORREF PALETTEINDEX(*wPaletteIndex*)
WORD *wPaletteIndex*; /* index to palette entry */

The **PALETTEINDEX** macro accepts an index to a logical-color palette entry and returns a value consisting of 1 in the high-order byte and the palette-entry index in the low-order byte. This is called a palette-entry specifier. An application using a color palette can pass this specifier instead of an explicit RGB value to functions that expect a color. This allows the function to use the color in the specified palette entry.

Parameters

wPaletteIndex
Specifies an index to the palette entry containing the color to be used for a graphics operation.

Return Value

The return value is a logical-palette index specifier. When using a logical palette, an application can use this specifier in place of an explicit RGB value for graphics-device interface (GDI) functions that require a color.

Comments

The **PALETTEINDEX** macro is defined in WINDOWS.H as follows:

```
#define PALETTEINDEX(i)  ((COLORREF)(0x01000000L | (DWORD)(WORD)(i)))
```

See Also

PALETTERGB, RGB

PALETTERGB

3.0

COLORREF PALETTERGB(*cRed*, *cGreen*, *cBlue*)

```
BYTE cRed;      /* red component of palette-relative RGB     */
BYTE cGreen;    /* green component of palette-relative RGB   */
BYTE cBlue;     /* blue component of palette-relative RGB    */
```

The **PALETTERGB** macro accepts three values representing relative intensities of red, green, and blue and returns a value consisting of 2 in the high-order byte and an RGB value in the three low-order bytes. This is called a palette-relative RGB specifier. An application using a color palette can pass this specifier instead of an explicit RGB value to functions that expect a color.

For output devices that support logical palettes, Windows matches a palette-relative RGB value to the nearest color in the logical palette of the device context as though the application had specified an index to that palette entry. If an output device does not support a system palette, then Windows uses the palette-relative RGB as though it were a conventional RGB doubleword returned by the **RGB** macro.

Parameters

cRed
Specifies the intensity of the red color field.

cGreen
Specifies the intensity of the green color field.

cBlue
Specifies the intensity of the blue color field.

Return Value

The return value specifies a palette-relative RGB value.

Comments The **PALETTERGB** macro is defined in WINDOWS.H as follows:

```
#define PALETTERGB(r,g,b)   (0x02000000L | RGB(r,g,b))
```

See Also **PALETTEINDEX, RGB**

RGB

2.x

COLORREF RGB(*cRed*, *cGreen*, *cBlue*)
BYTE *cRed*; /* red component of color */
BYTE *cGreen*; /* green component of color */
BYTE *cBlue*; /* blue component of color */

The **RGB** macro selects an RGB color based on the parameters supplied and the color capabilities of the output device.

Parameters

cRed
Specifies the intensity of the red color field.

cGreen
Specifies the intensity of the green color field.

cBlue
Specifies the intensity of the blue color field.

Return Value The return value specifies the resultant RGB color.

Comments The intensity for each argument can range from 0 through 255. If all three intensities are specified as zero, the result is black. If all three intensities are specified as 255, the result is white.

For information on using color values in a color palette, see the descriptions of the **PALETTEINDEX** and **PALETTERGB** macros earlier in this chapter.

Comments The **RGB** macro is defined in WINDOWS.H as follows:

```
#define RGB(r,g,b)   ((COLORREF)(((BYTE)(r)|((WORD)(g)<<8))| \
    (((DWORD)(BYTE)(b))<<16)))
```

See Also **GetBValue, GetGValue, GetRValue, PALETTEINDEX, PALETTERGB**

SELECTOROF

3.1

WORD SELECTOROF(*lp*)
void FAR* *lp*; /* long pointer */

The **SELECTOROF** macro retrieves the segment selector from the specified long pointer.

Parameters

lp
Specifies a long pointer.

Return Value

The return value is the segment selector.

Comments

The **SELECTOROF** macro is defined in WINDOWS.H as follows:

```
#define SELECTOROF(lp)      HIWORD(lp)
```

See Also

HIWORD, OFFSETOF

UnlockData

2.x

HANDLE UnlockData(*dummy*)

The **UnlockData** macro unlocks the current data segment. It is intended to be used by modules that have movable data segments.

Parameters

dummy
This parameter is ignored.

Return Value

The return value specifies the outcome of the **UnlockSegment** function. It is zero if the segment's lock count was decreased to zero. Otherwise, the return value is nonzero.

Comments

The **UnlockData** macro is defined in WINDOWS.H as follows:

```
#define UnlockData(dummy)   UnlockSegment((UINT)-1)
```

See Also

LockData, UnlockSegment

UnlockResource

2.x

BOOL UnlockResource(*hResData*)
HGLOBAL *hResData*; /* handle of memory object to unlock */

The **UnlockResource** macro unlocks the resource specified by the *hResData* parameter and decreases the reference count of the resource by one.

Parameters

hResData
Identifies the global memory object to be unlocked.

Return Value

The return value is zero if the object's reference count is decreased to zero. Otherwise, it is nonzero.

Comments

The **UnlockResource** macro is defined in WINDOWS.H as follows:

```
#define UnlockResource(h)   GlobalUnlock(h)
```

See Also

GlobalUnlock

Printer Escapes

Chapter 5

This chapter contains an alphabetic list of printer escapes for the Microsoft Windows operating system, version 3.1. The printer escapes allow applications to access certain facilities of output devices that are not directly available through the graphics device interface (GDI). The escape calls are made by an application, translated by Windows, and then sent to the printer driver.

ABORTDOC

short Escape(*hdc*, ABORTDOC, NULL, NULL, NULL)

The **ABORTDOC** printer escape is maintained for backwards compatibility. Applications written for Windows 3.1 should use the **AbortDoc** function.

This escape stops the current job and erases everything the application has written to the device since the last **ENDDOC** escape.

The **ABORTDOC** escape should be used to stop:

- Printing operations that do not specify an Abort function by using the **SETABORTPROC** escape.
- Printing operations that have not yet reached their first call to the **NEWFRAME** or **NEXTBAND** escape.

Parameters

hdc
HDC Identifies the device context.

Return Value

This escape does not return a value.

Comments

If an application encounters a printing error, it should not try to stop the operation by using the **Escape** function with either the **ENDDOC** or **ABORTDOC** escape. Graphics device interface (GDI) automatically terminates the operation before returning the error value.

If the application displays a dialog box to allow the user to cancel the print operation, it must send the **ABORTDOC** escape before destroying the dialog box.

The application must send the **ABORTDOC** escape before freeing the procedure-instance address of the Abort function, if any.

See Also

Escape

BANDINFO

short Escape(*hdc*, BANDINFO, sizeof(BANDINFOSTRUCT), *lpInData*, *lpOutData*)

The **BANDINFO** printer escape is maintained for backwards compatibility. Applications written for Windows 3.1 should send both text and graphics in every band.

This escape copies information about a device with banding capabilities to a structure pointed to by the *lpOutData* parameter. It is implemented only for devices that use banding to send output to the printer.

Banding is the property of an output device that allows a page of output to be stored in a metafile and divided into bands, each of which is sent to the device to create a complete page.

The information copied to the structure pointed to by the *lpOutData* parameter includes:

- A value that indicates whether there are graphics in the next band.
- A value that indicates whether there is text on the page.
- A **RECT** structure that contains a bounding rectangle for all graphics on the page.

If no data is returned, the *lpOutData* parameter is NULL.

The *lpInData* parameter specifies information sent by the application to the printer driver. This information is read by the driver only on the first call to the **BANDINFO** escape on a page.

Parameters

hdc
HDC Identifies the device context.

lpInData
BANDINFOSTRUCT FAR * Points to a **BANDINFOSTRUCT** structure that contains information to be passed to the driver. For more information about this structure, see the following Comments section.

lpOutData
BANDINFOSTRUCT FAR * Points to a **BANDINFOSTRUCT** structure that contains information returned by the driver. For more information about this structure, see the following Comments section.

Return Value

The return value specifies the outcome of the escape. This value is 1 if the escape is successful. It is zero if the function fails or is not implemented by the driver.

Comments

The **BANDINFOSTRUCT** structure contains information about the contents of a page and supplies a bounding rectangle for graphics on the page. This structure has the following form:

```
typedef struct tagBANDINFOSTRUCT {
    BOOL    fGraphics;
    BOOL    fText;
    RECT    rcGraphics;
} BANDINFOSTRUCT;
```

Following are the members in the **BANDINFOSTRUCT** structure:

fGraphics
Specifies nonzero if graphics are or are expected to be on the page or in the band. Otherwise, it is zero.

fText
Specifies nonzero if text is or is expected to be on the page or in the band. Otherwise, it is zero.

rcGraphics
Contains a **RECT** structure that supplies a bounding region for all graphics on the page.

The meaning of these members depends on which parameter contains the structure, as follows.

Member	When used in *lpInData*	When used in *lpOutData*
fGraphics	Nonzero if the application informs the driver that graphics are on the page	Nonzero if the driver informs the application that it expects graphics in this band
fText	Nonzero if the application informs the driver that text is on the page	Nonzero if the driver informs the application that it expects text in this band
rcGraphics	Bounding rectangle supplied for all graphics on the page	No valid return data

An application should call this escape immediately after each call to the **NEXTBAND** escape. The **BANDINFO** escape is in reference to the band that the driver returned to the **NEXTBAND** escape.

An application should use this escape in the following manner:

- On the first band, the driver may give the application a full-page band and ask for text only (the **fGraphics** member is set to zero and the **fText** member is set to nonzero). Then the application sends only text to the driver.

- If in the first band the application indicates that it has graphics (the **fGraphics** member is set to nonzero) or the driver encounters vector fonts, the driver bands the rest of the page.
- If there are no graphics or vector fonts, the next **NEXTBAND** escape returns an empty rectangle to indicate that the application should move on to the next page.
- If there are graphics but no vector fonts (the application sets the **fGraphics** member to nonzero, but there are no graphics in the first full-page text band), the driver may optionally band only into the rectangle the application passes for subsequent bands. This rectangle bounds all graphics on the page.
- If there are vector fonts, the driver bands the entire width and depth of the page with the **fText** member set to nonzero. It also sets the **fGraphics** flag to nonzero if the application has set it.

The driver assumes that an application using the **BANDINFO** escape only sends text in the first full-page text band because that is all the driver has requested. Therefore, if the driver encounters a vector font or graphics in the band, it assumes they were generated by a text primitive and sets the **fText** member to nonzero for all subsequent graphics bands, so they can be output as graphics. If the application does not meet this expectation, the image still generates properly, but the driver spends time sending spurious text primitives to graphics bands.

Older drivers written before the **BANDINFO** escape was designed use full-page banding for text. If a particular driver does not support the **BANDINFO** escape but sets the RC_BANDING raster capability, the application can detect full-page banding for text by determining if the first band on the page covers the entire page.

BEGIN_PATH

short Escape(*hdc***, BEGIN_PATH, NULL, NULL, NULL)**

The **BEGIN_PATH** printer escape opens a path. A path is a connected sequence of primitives drawn in succession to form a single polyline or polygon. Paths enable applications to draw complex borders, filled shapes, and clipping regions by supplying a collection of other primitives to define the desired shape.

Printer escapes supporting paths enable applications to render images on sophisticated devices, such as PostScript printers, without generating huge polygons to simulate the images.

To draw a path, an application first issues the **BEGIN_PATH** escape. Then it draws the primitives defining the border of the desired shape and issues an **END_PATH** escape, which includes a parameter specifying how the path is to be rendered.

Parameters

hdc
HDC Identifies the device context.

Return Value

The return value specifies the current path nesting level. This value is the number of calls to the **BEGIN_PATH** escape without a corresponding call to the **END_PATH** escape if the escape is successful. Otherwise, the return value is zero.

Comments

This escape is used only by PostScript printer drivers.

An application may begin a subpath within another path. If the subpath is closed, it is treated just like a polygon. If it is open, it is treated just like a polyline.

An application may use the **CLIP_TO_PATH** escape to define a clipping region corresponding to the interior or exterior of the currently open path.

CLIP_TO_PATH

short Escape(*hdc***, CLIP_TO_PATH, sizeof(int),** *lpClipMode***, NULL)**

The **CLIP_TO_PATH** printer escape defines a clipping region bounded by the currently open path. It enables the application to save and restore the current clipping region and to set up an inclusive or exclusive clipping region bounded by the currently open path. If the path defines an inclusive clipping region, portions of primitives falling outside the interior bounded by the path are clipped. If the path defines an exclusive clipping region, portions of primitives falling inside the interior are clipped.

Parameters

hdc

HDC Identifies the device context.

lpClipMode

LPINT Points to a short integer that specifies the clipping mode. It can be one of the following values:

Value	Meaning
CLIP_SAVE (0)	Saves the current clipping region.
CLIP_RESTORE (1)	Restores the previous clipping region.
CLIP_INCLUSIVE (2)	Sets an inclusive clipping region.
CLIP_EXCLUSIVE (3)	Sets an exclusive clipping region.

Return Value

The return value specifies the outcome of the escape. This value is nonzero if the escape is successful. Otherwise, it is zero.

Comments

This escape is used only by PostScript printer drivers.

To clip a set of primitives against a path, an application should follow these steps:

1. Save the current clipping region by using the **CLIP_TO_PATH** escape.
2. Begin a path with the **BEGIN_PATH** escape.
3. Draw the primitives bounding the clipping region.
4. Close the path with the **END_PATH** escape.
5. Set the clipping region by using the **CLIP_TO_PATH** escape.
6. Draw the primitives to be clipped.
7. Restore the original clipping region by using the **CLIP_TO_PATH** escape.

DEVICEDATA

short Escape(*hdc***, DEVICEDATA,** *nCount***,** *lpInData***,** *lpOutData***)**

The **DEVICEDATA** printer escape is identical to the **PASSTHROUGH** escape. For further information, see the description of **PASSTHROUGH**.

DRAFTMODE

short Escape(*hdc*, DRAFTMODE, sizeof(int), *lpDraftMode*, NULL)

The **DRAFTMODE** printer escape turns draft mode off or on. Turning draft mode on instructs the driver to print faster and with lower quality, if necessary. The draft mode can be changed only at page boundaries (for example, after a **NEWFRAME** escape directing the driver to advance to a new page).

Parameters

hdc
HDC Identifies the device context.

lpDraftMode
LPINT Points to a short integer that specifies the draft mode. It can be one of the following values:

Value	Meaning
0	Specifies draft mode off.
1	Specifies draft mode on.

Return Value

The return value specifies the outcome of the escape. This value is positive if the escape is successful. Otherwise, it is zero or negative.

Comments

The default draft mode is off.

DRAWPATTERNRECT

short Escape(*hdc*, DRAWPATTERNRECT, sizeof(PRECTSTRUCT), *lpInData*, NULL)

The **DRAWPATTERNRECT** printer escape creates a pattern, gray scale, or solid black rectangle by using the pattern and rule capabilities of Page Control Language (PCL) on Hewlett-Packard LaserJet or LaserJet-compatible printers. A gray scale is a gray pattern that contains a specific mixture of black and white pixels.

Parameters

hdc
HDC Identifies the device context.

lpInData
PRECT_STRUCT FAR * Points to a **PRECT_STRUCT** structure that describes the rectangle. For more information on this structure, see the following Comments section.

Return Value

The return value specifies the outcome of the escape. This value is 1 if the escape is successful. Otherwise, it is zero.

Comments

The *lpInData* parameter points to a **PRECT_STRUCT** structure that defines the rectangle to be created. This structure has the following form:

```
struct PRECT_STRUCT {
    POINT ptPosition;
    POINT ptSize;
    WORD  wStyle;
    WORD  wPattern;
};
```

Following are the members in the **PRECT_STRUCT** structure:

ptPosition
Specifies the upper-left corner of the rectangle.

ptSize
Specifies the lower-right corner of the rectangle.

wStyle
Specifies the type of pattern. It can be one of the following values:

Value	Meaning
0	Black rule
1	White rule that erases bitmap data previously written to same area (available on the HP LaserJet IIP only)
2	Gray scale
3	HP-defined

wPattern
Specifies the pattern. It is ignored for a black rule. It specifies the percentage of gray for a gray-scale pattern. It represents one of six patterns defined by Hewlett-Packard.

Comments

The output of the **DRAWPATTERNRECT** escape does not go through the graphics banding bitmap; it is sent to the printer in the text band. An application can use this escape to print line and block graphics without using graphics banding at all. Because many applications use only horizontal and vertical lines or blocks in graphic output, this is a significant optimization.

An application should use the **QUERYESCSUPPORT** escape to determine whether a device is capable of drawing patterns and rules before using the **DRAWPATTERNRECT** escape. If an application uses the **BANDINFO** escape, all patterns and rectangles sent by using **DRAWPATTERNRECT** should be treated as text and sent on a text band.

Applications that use the **DRAWPATTERNRECT** escape must observe two limitations. First, rules drawn with **DRAWPATTERNRECT** are not subject to clipping regions in the device context. Second, applications should not try to erase patterns and rules created with **DRAWPATTERNRECT** by placing opaque objects over them. If the printer supports white rules, these can be used to erase patterns created by **DRAWPATTERNRECT**. If the printer does not support white rules, there is no method for erasing these patterns.

If an application cannot use the **DRAWPATTERNRECT** escape, it should generally use the **PatBlt** function instead. (If **PatBlt** is used to print a black rectangle, the application should use the BLACKNESS raster operator.) If the device is a plotter, the application should use the **Rectangle** function.

ENABLEDUPLEX

short Escape(*hdc***, ENABLEDUPLEX, sizeof(WORD),** *lpInData***, NULL)**

The **ENABLEDUPLEX** printer escape is maintained for backwards compatibility. Applications written for Windows 3.1 should use the **ExtDeviceMode** function. An application can determine whether an output device is capable of creating duplex output by checking the DM_DUPLEX bit of the **dmFields** member in the **DEVMODE** structure.

This escape enables the duplex printing capabilities of a printer. A device that possesses duplex printing capabilities is able to print on both sides of the output medium.

Parameters

hdc

HDC Identifies the device context.

lpInData

LPWORD Points to an unsigned 16-bit integer that specifies whether duplex or simplex printing is used. It can be one of the following values:

Value	Meaning
0	Simplex
1	Duplex with vertical binding
2	Duplex with horizontal binding

Return Value

The return value specifies the outcome of the escape. This value is 1 if the escape is successful. Otherwise, it is zero.

Comments

An application should use the **QUERYESCSUPPORT** escape to determine whether an output device is capable of creating duplex output. If **QUERYESCSUPPORT** returns a nonzero value, the application should send the **ENABLEDUPLEX** escape even if simplex printing is desired. This procedure guarantees replacement of any values set in the driver-specific dialog box. If duplex printing is enabled and an uneven number of **NEXTFRAME** escapes are sent to the driver prior to the **ENDDOC** escape, the driver ejects an additional page before ending the print job.

ENABLEPAIRKERNING

short Escape(*hdc***, ENABLEPAIRKERNING, sizeof(int),** *lpNewKernFlag***,** *lpOldKernFlag***)**

The **ENABLEPAIRKERNING** printer escape enables or disables the ability of the driver to kern character pairs automatically. Kerning is the process of adding or subtracting space between characters in a string of text.

When pair kerning is enabled, the driver automatically kerns those pairs of characters that are listed in the character-pair kerning table for the font. The driver reflects this kerning both on the printer and in the **GetTextExtent** function calls.

Parameters

hdc
HDC Identifies the device context.

lpNewKernFlag
LPINT Points to a short-integer value that specifies whether automatic pair kerning is to be enabled (1) or disabled (zero).

lpOldKernFlag
LPINT Points to a short-integer value that receives the previous automatic pair-kerning value.

Return Value

The return value specifies the outcome of the escape. This value is 1 if the escape is successful. It is zero if the escape is not successful or not implemented.

Comments

The default state of this escape is zero; automatic character-pair kerning is disabled.

A driver does not have to support the **ENABLEPAIRKERNING** escape just because it supplies the character-pair kerning table to the application by using the **GETPAIRKERNTABLE** escape. When the **GETPAIRKERNTABLE** escape is supported but the **ENABLEPAIRKERNING** escape is not, the application must properly space the kerned characters on the output device by using the **ExtTextOut** function.

ENABLERELATIVEWIDTHS

short Escape(*hdc***, ENABLERELATIVEWIDTHS, sizeof(int),** *lpNewWidthFlag***,** *lpOldWidthFlag***)**

The **ENABLERELATIVEWIDTHS** printer escape enables or disables relative character widths. When relative widths are disabled (the default), the width of each character can be expressed as a number of device units. This method guarantees that the extent of a string will equal the sum of the extents of the characters in the string. This allows applications to build an extent table by using one-character **GetTextExtent** function calls.

When relative widths are enabled, the sum of a string may not equal the sum of the widths of the characters. Applications that enable this feature are expected to retrieve the extent table for the font and compute relatively scaled string widths.

Parameters

hdc
HDC Identifies the device context.

lpNewWidthFlag
LPINT Points to a short integer that specifies whether relative widths are to be enabled (1) or disabled (zero).

lpOldWidthFlag
LPINT Points to a short integer that receives the previous relative character width value.

Return Value

The return value specifies the outcome of the escape. This value is 1 if the escape is successful. It is zero if the escape is not successful or not implemented.

Comments

The default state of this escape is zero; relative character widths are disabled.

When the **ENABLERELATIVEWIDTHS** escape is enabled, the values specified as font units and accepted and returned by the escapes described in this chapter are returned in the relative units of the font.

It is assumed that only linear-scaling devices are dealt with in a relative mode. Nonlinear-scaling devices do not implement this escape.

ENDDOC

short Escape(*hdc*, ENDDOC, NULL, NULL, NULL)

The **ENDDOC** printer escape is maintained for backwards compatibility. Applications written for Windows 3.1 should use the **EndDoc** function.

This escape ends a print job started by a **STARTDOC** escape.

Parameters

hdc
HDC Identifies the device context.

Return Value

The return value specifies the outcome of the escape. This value is positive if the escape is successful. Otherwise, it is zero or negative.

Comments

The **ENDDOC** escape should not be used inside metafiles.

END_PATH

short Escape(*hdc*, END_PATH, sizeof(PATH_INFO), *lpInData*, NULL)

The **END_PATH** printer escape ends a path. A path is a connected sequence of primitives drawn in succession to form a single polyline or polygon. Paths enable applications to draw complex borders, filled shapes, and clipping regions by supplying a collection of other primitives to define the desired shape.

Printer escapes that support paths enable applications to render images on sophisticated devices, such as PostScript printers, without generating huge polygons to simulate them.

To draw a path, an application first issues the **BEGIN_PATH** escape. Then it draws the primitives defining the border of the desired shape and issues an **END_PATH** escape.

The **END_PATH** escape takes, as a parameter, a pointer to a structure specifying the manner in which the path is to be rendered. The structure specifies whether or not the path is to be drawn and whether it is open or closed. Open paths define polylines, and closed paths define fillable polygons.

Parameters

hdc
HDC Identifies the device context.

lpInData
PATH_INFO FAR * Points to a **PATH_INFO** structure that defines how the path is to be rendered. For more information about this structure, see the following Comments section.

Return Value

The return value specifies the current path nesting level. This value is the number of **BEGIN_PATH** escape calls without a corresponding **END_PATH** call if the escape is successful. Otherwise, it is –1.

Comments

This escape is used only by PostScript printer drivers.

An application may begin a subpath within another path. If the subpath is closed, it is treated just like a polygon. If it is open, it is treated just like a polyline.

An application may use the **CLIP_TO_PATH** escape to define a clipping region corresponding to the interior or exterior of the currently open path.

The *lpInData* parameter points to a **PATH_INFO** structure that specifies how to render the path. This structure has the following form:

```
struct PATH_INFO {
    short    RenderMode;
    BYTE     FillMode;
    BYTE     BkMode;
    LOGPEN   Pen;
    LOGBRUSH Brush;
    DWORD    BkColor;
};
```

Following are the members in the **PATH_INFO** structure:

RenderMode
Specifies how the path is to be rendered. It can be one of the following values:

Value	Meaning
NO_DISPLAY (0)	Path is not drawn.
OPEN (1)	Path is drawn as an open polygon.
CLOSED (2)	Path is drawn as a closed polygon.

FillMode
Specifies how the path is to be filled. It can be one of the following values:

Value	Meaning
ALTERNATE (1)	Fill is done using the alternate fill algorithm.
WINDING (2)	Fill is done using the winding fill algorithm.

BkMode
Specifies the background mode for filling the path. It can be one of the following values:

Value	Meaning
OPAQUE	Background is filled with the background color before the brush is drawn.
TRANSPARENT	Background is not changed.

Pen
Specifies the pen with which the path is to be drawn. If the **RenderMode** function is set to the NO_DISPLAY value, the pen is ignored.

Brush
Specifies the brush with which the path is to be filled. If the **RenderMode** function is set to the NO_DISPLAY or OPEN value, the brush is ignored.

BkColor
Specifies the color with which the path is filled if the **BkMode** function is set to the OPAQUE value.

ENUMPAPERBINS

short Escape(*hdc***, ENUMPAPERBINS, sizeof(int),** *lpNumBins***,** *lpOutData***)**

The **ENUMPAPERBINS** printer escape is maintained for backwards compatibility. Applications written for Windows 3.1 should call the **DeviceCapabilities** function with the DC_BINNAMES index to retrieve the number of available paper bins and the name of each bin.

This escape retrieves attribute information about a specified number of paper bins. The **GETSETPAPERBINS** escape retrieves the number of bins available on a printer.

Parameters

hdc
HDC Identifies the device context.

lpNumBins
LPINT Points to an integer that specifies the number of bins for which information is to be retrieved.

lpOutData
LPSTR Points to a structure to which information about the paper bins is copied. The size of the structure depends on the number of bins for which infor-

mation was requested. For a description of this structure, see the following Comments section.

Return Value

The return value specifies the outcome of the escape. This value is 1 if the escape is successful. It is zero if the escape is not successful or not implemented.

Comments

The structure to which the *lpOutData* parameter points consists of two arrays. The first is an array of short integers containing the paper-bin identifier numbers in the following form:

```
short BinList[cBinMax]
```

The number of integers in the array (the *cBinMax* value) is equal to the value pointed to by the *lpNumBins* parameter.

The second array in the structure to which *lpOutData* points is an array of characters in the following form:

```
char PaperNames[cBinMax][cchBinName]
```

The *cBinMax* value is equal to the value pointed to by the *lpNumBins* parameter. The *cchBinName* value is the length of each string (currently 24).

ENUMPAPERMETRICS

short Escape(*hdc***, ENUMPAPERMETRICS, sizeof(int),** *lpMode***,** *lpOutData***)**

The **ENUMPAPERMETRICS** printer escape performs one of two functions according to the mode:

- It determines the number of paper types supported and returns this value, which can then be used to allocate an array of **RECT** structures.
- It returns one or more **RECT** structures that define the areas on the page that can receive an image.

This escape is provided only for backward compatibility. An application should call the **DeviceCapabilities** function with the DC_PAPERSIZE index to discover the number of available paper sizes and the dimensions of each size.

Parameters

hdc

HDC Identifies the device context.

lpMode

LPINT Points to an integer that specifies the mode for the escape. It can be one of the following values:

Value	Meaning
0	Return value indicates how many **RECT** structures are required to contain the information about the available paper types.
1	Array of **RECT** structures to which the *lpOutData* parameter points is filled with the information.

lpOutData

LPRECT Points to an array of **RECT** structures that return all the areas capable of receiving an image.

Return Value

The return value is positive if the escape is successful. The value is zero if the escape is not implemented and negative if an error occurred.

EPSPRINTING

short Escape(*hdc*, EPSPRINTING, sizeof(BOOL), *lpBool*, NULL)

The **EPSPRINTING** printer escape suppresses the output of the Windows PostScript header control section, which is about 7K. If an application uses this escape, no graphics device interface (GDI) calls are allowed.

Parameters

hdc

HDC Identifies the device context.

lpBool

BOOL FAR * Points to a Boolean value that indicates whether downloading should be enabled (nonzero) or disabled (zero).

Return Value

The return value is positive if the escape is successful. This value is zero if the escape is not implemented and negative if an error occurred.

Comments

This escape is used only by PostScript printer drivers.

EXT_DEVICE_CAPS

short Escape(*hdc***, EXT_DEVICE_CAPS, sizeof(int),** *lpIndex***,** *lpCaps***)**

The **EXT_DEVICE_CAPS** printer escape retrieves information about device-specific capabilities. It supplements the **GetDeviceCaps** function.

Parameters

hdc

HDC Identifies the device context.

lpIndex

LPINT Points to a short integer that specifies the index of the capability to be retrieved. It can be any one of the following values:

Value	Meaning
R2_CAPS (1)	The *lpCaps* parameter indicates which of the 16 binary raster operations the device driver supports. A bit will be set for each supported raster operation. For further information, see the description of the **SetROP2** function in the *Microsoft Windows Programmer's Reference, Volume 2*.
PATTERN_CAPS (2)	The *lpCaps* parameter returns the maximum dimensions of a pattern brush bitmap. The low-order word of the capability value contains the maximum width of a pattern brush bitmap, and the high-order word contains the maximum height.
PATH_CAPS (3)	The *lpCaps* parameter indicates whether the device is capable of creating paths by using alternate and winding interiors, and whether the device can do exclusive or inclusive clipping to path interiors. The path capabilities are obtained by using the logical OR operation on the following values: PATH_ALTERNATE (1) PATH_WINDING (2) PATH_INCLUSIVE (4) PATH_EXCLUSIVE (8)
POLYGON_CAPS (4)	The *lpCaps* parameter returns the maximum number of polygon points supported by the device. The capability value is an unsigned value specifying the maximum number of points.

Value	Meaning
PATTERN_COLOR_CAPS (5)	The *lpCaps* parameter indicates whether the device can convert monochrome pattern bitmaps to color. The capability value is 1 if the device can do pattern bitmap color conversions and zero if it cannot.
R2_TEXT_CAPS (6)	The *lpCaps* parameter indicates whether the device is capable of performing binary raster operations on text. The low-order word of the capability value specifies which raster operations are supported for text. A bit is set for each supported raster operation, as in the R2_CAPS escape. The high-order word specifies the type of text to which the raster capabilities apply. It is obtained by applying the logical OR operation to the following values together: RASTER_TEXT (1) DEVICE_TEXT (2) VECTOR_TEXT (4)
POLYMODE_CAPS (7)	The *lpcaps* parameter indicates which poly modes are supported by the printer driver. The capability value is obtained by using the bitwise OR operator to combine a bit in the corresponding position for each supported poly mode. For example, if the printer supports the PM_POLYSCANLINE and PM_BEZIER poly modes, the capability value would be: (1 << PM_POLYSCANLINE) | (PM_BEZIER)

lpCaps
LPDWORD Points to a 32-bit integer to which the capabilities will be copied.

Return Value

The return value is nonzero if the specified extended capability is supported. This value is zero if the capability is not supported.

Comments

This escape is used only by PostScript printer drivers.

EXTTEXTOUT

short Escape(*hdc*, EXTTEXTOUT, sizeof(EXTTEXT_STRUCT), *lpInData*, NULL)

The **EXTTEXTOUT** printer escape provides an efficient way for an application to call the graphics device interface (GDI) **TextOut** function when justification, letter spacing, or kerning is involved.

This function is provided only for backward compatibility. New applications should use the GDI **ExtTextOut** function instead.

Parameters

hdc
HDC Identifies the device context.

lpInData
EXTTEXT_STRUCT FAR * Points to an **EXTTEXT_STRUCT** structure that specifies the initial position, characters, and character widths of the string. For more information about this structure, see the following Comments section.

Return Value

The return value specifies the outcome of the escape. This value is 1 if the escape is successful. It is zero if the escape is not successful or not implemented.

Comments

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

```
struct EXTTEXT_STRUCT {
    WORD   x;
    WORD   y;
    LPWORD lpText;
    LPWORD lpWidths;
};
```

Following are the members in the **EXTTEXT_STRUCT** structure:

x
Specifies the x-coordinate of the upper-left corner of the string's starting point.

y
Specifies the y-coordinate of the upper-left corner of the string's starting point.

lpText
Points to an array of *cch* character codes, where *cch* is the number of bytes in the string (*cch* is also the number of words in the width array).

lpWidths
Points to an array of *cch* character widths to use when printing the string. The first character appears at (**x**,**y**), the second at (**x** + **lpWidths**[0],**y**), the third at (**x** + **lpWidths**[0] + **lpWidths**[1],**y**), and so on. These character widths are specified in the font units of the currently selected font. (The character widths are always equal to device units, unless the application has enabled relative character widths.)

The units contained in the width array are specified as font units of the device.

FLUSHOUTPUT

short Escape(*hdc*, FLUSHOUTPUT, NULL, NULL, NULL)

The **FLUSHOUTPUT** printer escape clears all output from the device's buffer.

Parameters

hdc
HDC Identifies the device context.

Return Value

The return value specifies the outcome of the escape. This value is greater than zero if the escape is successful. Otherwise, it is less than zero.

GETCOLORTABLE

short Escape(*hdc*, GETCOLORTABLE, sizeof(int), *lpIndex*, *lpColor*)

The **GETCOLORTABLE** printer escape retrieves an RGB color-table entry and copies it to the location specified by the *lpColor* parameter.

Parameters

hdc
HDC Identifies the device context.

lpIndex
LPINT Points to a short integer that specifies the index of a color-table entry. Color-table indexes start at zero for the first table entry.

lpColor
LPDWORD Points to the long integer that will receive the RGB color value for the given entry.

Return Value

The return value specifies the outcome of the escape. This value is positive if the escape is successful. Otherwise, it is negative.

GETEXTENDEDTEXTMETRICS

short Escape(*hdc***, GETEXTENDEDTEXTMETRICS, sizeof(WORD),** *lpInData***,** *lpOutData***)**

The **GETEXTENDEDTEXTMETRICS** printer escape fills the buffer pointed to by the *lpOutData* parameter with the extended text metrics for the selected font.

Parameters

hdc
HDC Identifies the device context.

lpInData
LPWORD Points to an unsigned 16-bit integer that specifies the number of bytes pointed to by the *lpOutData* parameter.

lpOutData
EXTTEXTMETRIC FAR * Points to an **EXTTEXTMETRIC** structure. For more information about this structure, see the following Comments section.

Return Value

The return value specifies the number of bytes copied to the buffer pointed to by the *lpOutData* parameter. This value will never exceed that specified in the *nSize* member pointed to by the *lpInData* parameter. The return value is zero if the selected font does not have the extended text metrics or if the escape fails or is not implemented.

Comments

The *lpOutData* parameter points to an **EXTTEXTMETRIC** structure, which has the following form:

```
struct EXTTEXTMETRIC {
    short etmSize;
    short etmPointSize;
    short etmOrientation;
    short etmMasterHeight;
    short etmMinScale;
    short etmMaxScale;
    short etmMasterUnits;
    short etmCapHeight;
    short etmXHeight;
    short etmLowerCaseAscent;
    short etmLowerCaseDescent;
    short etmSlant;
    short etmSuperScript;
    short etmSubScript;
    short etmSuperScriptSize;
    short etmSubScriptSize;
    short etmUnderlineOffset;
    short etmUnderlineWidth;
    short etmDoubleUpperUnderlineOffset;
    short etmDoubleLowerUnderlineOffset;
    short etmDoubleUpperUnderlineWidth;
    short etmDoubleLowerUnderlineWidth;
    short etmStrikeOutOffset;
    short etmStrikeOutWidth;
    WORD  etmKernPairs;
    WORD  etmKernTracks;
};
```

Following are the members in the **EXTTEXTMETRIC** structure:

etmSize
Specifies the size of the structure, in bytes.

etmPointSize
Specifies the nominal point size of this font, in twips (1/20 of a point, or 1/1440 inch). This is the intended size of the font; the actual size may differ slightly depending on the resolution of the device.

etmOrientation
Specifies the orientation of the font. The **etmOrientation** member may be any of the following values:

Value	Meaning
0	Either orientation
1	Portrait
2	Landscape

These values refer to the ability of this font to be placed on a page with the given orientation. A portrait page has a height that is greater than its width. A landscape page has a width that is greater than its height.

etmMasterHeight
Specifies the font size, in device units, for which the values in this font's extent table are exact.

etmMinScale
Specifies the minimum valid size for this font. The following equation illustrates how the minimum point size is determined:

smallest point size = (**etmMinScale** * 72) / *dfVertRes*

The value 72 represents the number of points per inch. The *dfVertRes* value is the number of dots per inch.

etmMaxScale
Specifies the maximum valid size for this font. The following equation illustrates how the maximum point size is determined:

largest point size = (**etmMaxScale** * 72) / *dfVertRes*

The value 72 represents the number of points per inch. The *dfVertRes* value is the number of dots per inch.

etmMasterUnits
Specifies the integer number of units per em where an em equals the value of the **etmMasterHeight** member. (That is, **etmMasterUnits** is **emtMasterHeight** expressed in font units instead of device units.)

etmCapHeight
Specifies the height, in font units, of uppercase characters in the font. Typically, this is the height of capital *H*.

etmXHeight
Specifies the height, in font units, of lowercase characters in the font. Typically, this is the height of lowercase *x*.

etmLowerCaseAscent
Specifies the distance, in font units, that the ascender of lowercase letters extends above the base line. Typically, this is the height of lowercase *d*.

etmLowerCaseDescent
Specifies the distance, in font units, that the descender of lowercase letters extends below the base line. Typically, this is specified for the descender of lowercase *p*.

etmSlant
Specifies, for an italic or slanted font, the angle of the slant measured in tenths of a degree clockwise from the upright version of the font.

etmSuperScript
Specifies, in font units, the recommended amount to offset superscript characters from the base line. This is typically a negative value.

etmSubScript
Specifies, in font units, the recommended amount to offset subscript characters from the base line. This is typically a positive value.

etmSuperScriptSize
Specifies, in font units, the recommended size of superscript characters for this font.

etmSubScriptSize
Specifies, in font units, the recommended size of subscript characters for this font.

etmUnderlineOffset
Specifies, in font units, the offset downward from the base line where the top of a single underline bar should appear.

etmUnderlineWidth
Specifies, in font units, the thickness of the underline bar.

etmDoubleUpperUnderlineOffset
Specifies the offset, in font units, downward from the base line where the top of the upper double-underline bar should appear.

etmDoubleLowerUnderlineOffset
Specifies the offset, in font units, downward from the base line where the top of the lower double-underline bar should appear.

etmDoubleUpperUnderlineWidth
Specifies, in font units, the thickness of the upper underline bar.

etmDoubleLowerUnderlineWidth
Specifies, in font units, the thickness of the lower underline bar.

etmStrikeOutOffset
Specifies, in font units, the offset upward from the base line where the top of a strikeout bar should appear.

etmStrikeOutWidth
Specifies the thickness, in font units, of the strikeout bar.

etmKernPairs
Specifies the number of character kerning pairs defined for this font. An application can use this value to calculate the size of the pair-kern table returned by the **GETPAIRKERNTABLE** escape. It will not be greater than 512 kerning pairs.

etmKernTracks
Specifies the number of kerning tracks defined for this font. An application can use this value to calculate the size of the track-kern table returned by the **GETTRACKKERNTABLE** escape. It will not be greater than 16 kerning tracks.

The values returned in many of the members of the **EXTTEXTMETRIC** structure are affected by whether relative character widths are enabled or disabled. For more information, see the description of the **ENABLERELATIVEWIDTHS** escape earlier in this chapter.

GETEXTENTTABLE

short Escape(*hdc***, GETEXTENTTABLE, sizeof(CHAR_RANGE_STRUCT),** *lpInData*, *lpOutData*)

The **GETEXTENTTABLE** printer escape retrieves the width (extent) of individual characters from a group of consecutive characters in the character set for the selected font.

Parameters

hdc
HDC Identifies the device context.

lpInData
LPSTR Points to a **CHAR_RANGE_STRUCT** structure that defines the range of characters for which the width is to be retrieved. For more information about this structure, see the following Comments section.

lpOutData
LPINT Points to an array of short integers that receives the character widths. The size of the array must be at least (**chLast** – **chFirst** + 1).

Return Value

The return value specifies the outcome of the escape. This value is 1 if the escape is successful. It is zero if the escape is not successful. If the escape is not implemented, the return value is zero.

Comments

The *lpInData* parameter points to a **CHAR_RANGE_STRUCT** structure that defines the range of characters for which the width is to be retrieved. This structure has the following form:

```
struct CHAR_RANGE_STRUCT {
    CHAR chFirst;
    CHAR chLast;
};
```

Following are the members in the **CHAR_RANGE_STRUCT** structure:

chFirst
Specifies the character code of the first character whose width is to be retrieved.

chLast
Specifies the character code of the last character whose width is to be retrieved.

How an application uses the retrieved values depends upon whether relative character widths are enabled or disabled. For more information, see the description of the **ENABLERELATIVEWIDTHS** escape, earlier in this chapter.

GETFACENAME

short Escape(*hdc***, GETFACENAME, NULL, NULL,** *lpFaceName***)**

The **GETFACENAME** printer escape retrieves the face name of the current physical font.

Parameters

hdc
HDC Identifies the device context.

lpFaceName
LPSTR Points to a buffer of characters to receive the face name. This buffer must be at least 60 bytes in length.

Return Value

The return value is positive if the escape was successful. This value is zero if the escape is not implemented or negative if an error occurred.

Comments

This escape is used only by PostScript printer drivers.

GETPAIRKERNTABLE

short Escape(*hdc***, GETPAIRKERNTABLE, NULL, NULL,** *lpOutData***)**

The **GETPAIRKERNTABLE** printer escape fills the buffer pointed to by the *lpOutData* parameter with the character-pair kerning table for the selected font.

Parameters

hdc

HDC Identifies the device context.

lpOutData

KERNPAIR FAR * Points to an array of **KERNPAIR** structures. This array must be large enough to accommodate the entire character-pair kerning table for the font. The number of character-kerning pairs in the font can be obtained from the **EXTTEXTMETRIC** structure returned by the **GETEXTENDEDTEXTMETRICS** escape. For more information about this structure, see the following Comments section.

Return Value

The return value specifies the number of **KERNPAIR** structures copied to the buffer. This value is zero if the font does not have kerning pairs defined or the escape fails or is not implemented.

Comments

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

```
struct KERNPAIR {
    union {
           BYTE each  [2];  /* 'each' and 'both' share same memory */
           WORD both;
         } kpPair;
    short kpKernAmount;
};
```

Following are the members in the **KERNPAIR** structure:

each

Specifies the character codes for the kerning pair.

both

Specifies a 16-bit value in which the first character in the kerning pair is in the low-order byte and the second character is in the high-order byte.

kpKernAmount

Specifies the signed amount that this pair will be kerned if they appear side by side in the same font and size. This value is typically negative because pair-kerning usually results in two characters being set tighter than normal.

The array of **KERNPAIR** structures is sorted in increasing order by the **kpPair.both** member.

The values returned in **KERNPAIR** structures are affected by whether relative character widths are enabled or disabled. For more information, see the description of the **ENABLERELATIVEWIDTHS** escape earlier in this chapter.

GETPHYSPAGESIZE

short Escape(*hdc*, GETPHYSPAGESIZE, NULL, NULL, *lpDimensions*)

The **GETPHYSPAGESIZE** printer escape retrieves the physical page size and copies it to the location pointed to by the *lpDimensions* parameter.

Parameters

hdc
HDC Identifies the device context.

lpDimensions
LPPOINT Points to a **POINT** structure that will receive the physical page dimensions (in the current orientation). The **x** member of the **POINT** structure receives the horizontal size, in device units, and the **y** member receives the vertical size, in device units.

Return Value

The return value specifies the outcome of the escape. This value is positive if the escape is successful. Otherwise, it is zero or negative.

GETPRINTINGOFFSET

short Escape(*hdc*, GETPRINTINGOFFSET, NULL, NULL, *lpOffset*)

The **GETPRINTINGOFFSET** printer escape retrieves the offset from the upper-left corner of the physical page where the actual printing or drawing begins. This escape is generally not useful for devices that allow the user to set the origin of the printable area directly.

Parameters

hdc
HDC Identifies the device context.

lpOffset
LPPOINT Points to a **POINT** structure that will receive the printing offset (in the current orientation). The **x** member of the **POINT** structure receives the horizontal coordinate of the printing offset, in device units, and the **y** member receives the vertical coordinate of the printing offset, in device units.

Return Value

The return value specifies the outcome of the escape. This value is positive if the escape is successful. Otherwise, it is zero or negative.

GETSCALINGFACTOR

short Escape(*hdc*, **GETSCALINGFACTOR, NULL, NULL,** *lpFactors*)

The **GETSCALINGFACTOR** printer escape retrieves the scaling factors for the x-axis and y-axis of a printing device. For each scaling factor, the escape copies an exponent of 2 to the location pointed to by the *lpFactors* parameter. For example, the value 3 is copied to *lpFactors* if the scaling factor is 8.

Scaling factors are used by printing devices that support graphics at a smaller resolution than text.

Parameters

hdc
HDC Identifies the device context.

lpFactors
LPPOINT Points to the **POINT** structure that will receive the scaling factor. The **x** member of the **POINT** structure receives the scaling factor for the x-axis and the **y** member receives the scaling factor for the y-axis.

Return Value

The return value specifies the outcome of the escape. This value is positive if the escape is successful. Otherwise, it is zero or negative.

GETSETPAPERBINS

short Escape(*hdc*, **GETSETPAPERBINS,** *nCount*, *lpInData*, *lpOutData*)

The **GETSETPAPERBINS** printer escape is maintained for backwards compatibility. Applications written for Windows 3.1 should call the **DeviceCapabilities** function with the DC_BINS index to retrieve the number of available paper bins and use the **ExtDeviceMode** function to set the current paper bin.

This escape retrieves the number of paper bins available on a printer and sets the current paper bin. For more information about actions performed by this escape, see the following Comments section.

Parameters

hdc
HDC Identifies the device context.

nCount
int Specifies the number of bytes pointed to by the *lpInData* parameter.

lpInData
BinInfo FAR * Points to a **BinInfo** structure that specifies the new paper bin. It may be set to NULL. For more information about this structure, see the following Comments section.

lpOutData
BinInfo FAR * Points to a **BinInfo** structure that contains information about the current or previous paper bin and the number of bins available. For more information about this structure, see the following comments section.

Return Value

The return value is positive if the escape is successful. Otherwise, this value is zero or negative.

Comments

There are three possible actions for this escape, depending on the values passed in the *lpInData* and *lpOutData* parameters:

lpInData	*lpOutData*	Action
NULL	**BinInfo**	Retrieves the number of bins and the number of the current bin.
BinInfo	**BinInfo**	Sets the current bin to the number specified in the **BinNumber** member of the structure to which the *lpInData* parameter points and retrieves the number of the previous bin.
BinInfo	NULL	Sets the current bin to the number specified in the **BinNumber** member of the structure to which the *lpInData* parameter points.

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

```
struct BinInfo {
    int BinNumber;
    int cBins;
    int Reserved;
    int Reserved;
    int Reserved;
    int Reserved;
};
```

Following are the members of the **BinInfo** structure:

BinNumber
Identifies the current or previous paper bin.

cBins
Specifies the number of paper bins available.

Once a new bin is set, the selection takes effect immediately; the next page printed comes from the new bin.

GETSETPAPERMETRICS

short Escape(*hdc***, GETSETPAPERMETRICS, sizeof(RECT),** *lpNewPaper***,** *lpPrevPaper***)**

The **GETSETPAPERMETRICS** printer escape sets the paper type according to the given paper metrics information. It also retrieves the paper metrics information for the current printer.

This escape is obsolete. Printer drivers written for Windows version 3.0 and later may not support this escape. Applications can use the **DeviceCapabilities** and **ExtDeviceMode** functions to achieve the same functionality.

This escape expects a **RECT** structure representing the imageable area of the physical page and assumes the origin is situated in the upper-left corner.

Parameters

hdc
HDC Identifies the device context.

lpNewPaper
LPRECT Points to a **RECT** structure that defines the new imageable area.

lpPrevPaper
LPRECT Points to a **RECT** structure that receives the previous imageable area.

Return Value

The return value is positive if the escape is successful. The value is zero if the escape is not implemented and negative if an error occurs.

GETSETPRINTORIENT

short Escape(*hdc***, GETSETPRINTORIENT,** *nCount***,** *lpInData***, NULL)**

The **GETSETPRINTORIENT** printer escape returns or sets the current paper orientation. This escape is obsolete. Printer drivers written for Windows version 3.0 and later may not support this escape. An application should call the **ExtDeviceMode** function instead.

Parameters

hdc
HDC Identifies the device context.

nCount
short Specifies the number of bytes pointed to by the *lpInData* parameter.

lpInData
ORIENT FAR * Points to an **ORIENT** structure that specifies the new paper orientation. For a description of this structure, see the following Comments section. It may be set to NULL, in which case the **GETSETPRINTORIENT** escape returns the current paper orientation.

Return Value

The return value specifies the current orientation if *lpInData* is NULL. Otherwise, this value is the previous orientation. This value is –1 if the escape fails.

Comments

This escape is provided only for backward compatibility. New applications should use the graphics device interface (GDI) **DeviceCapabilities** and **ExtDeviceMode** functions instead.

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

```
struct ORIENT {
    DWORD Orientation;
    DWORD Reserved;
    DWORD Reserved;
    DWORD Reserved;
    DWORD Reserved;
};
```

The **Orientation** member can be one of these values:

Value	Meaning
1	New orientation is portrait.
2	New orientation is landscape.

GETSETSCREENPARAMS

short Escape(*hdc***, GETSETSCREENPARAMS, sizeof(SCREENPARAMS),** *lpInData***,** *lpOutData***)**

The **GETSETSCREENPARAMS** printer escape retrieves or sets the current screen information for rendering halftones.

Parameters

hdc
HDC Identifies the device context.

lpInData
SCREENPARAMS FAR * Points to a **SCREENPARAMS** structure that contains the new screen information. For more information about this structure, see the following Comments section. This parameter may be NULL.

lpOutData
SCREENPARAMS FAR * Points to a **SCREENPARAMS** structure that retrieves the previous screen information. For more information about this structure, see the following Comments section. This parameter may be NULL.

Return Value

The return value specifies the outcome of the escape. This value is positive if the escape is successful. Otherwise, it is negative.

Comments

This escape affects how device-independent bitmaps (DIBs) are rendered and how color objects are filled.

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

```
typedef struct tagSCREENPARAMS {
  int   angle;
  int   frequency;
} SCREENPARAMS;
```

Following are the members of the **SCREENPARAMS** structure:

angle
Specifies, in degrees, the angle of the halftone screen.

frequency
Specifies, in dots per inch, the screen frequency.

GETTECHNOLOGY

short Escape(*hdc***, GETTECHNOLOGY, NULL, NULL, *lpTechnology*)**

The **GETTECHNOLOGY** printer escape retrieves the general technology type for a printer, which allows an application to perform technology-specific actions.

Applications should avoid using this escape. Printer drivers written for Windows version 3.0 and later may not support this escape.

Parameters

hdc
HDC Identifies the device context.

lpTechnology
LPSTR Points to a buffer to which the driver copies a null-terminated string containing the printer technology type, such as "PostScript".

Return Value

The return value specifies the outcome of the escape. This value is 1 if the escape is successful. It is zero if the escape is not successful or is not implemented.

GETTRACKKERNTABLE

short Escape(*hdc*, GETTRACKKERNTABLE, NULL, NULL, *lpOutData*)

The **GETTRACKKERNTABLE** printer escape fills the buffer pointed to by the *lpOutData* parameter with the track-kerning table for the currently selected font.

Parameters

hdc
HDC Identifies the device context.

lpOutdata
KERNTRACK FAR * Points to an array of **KERNTRACK** structures. This array must be large enough to accommodate all the kerning tracks for the font. The number of tracks in the font can be obtained from the **EXTTEXTMETRIC** structure which is returned by the **GETEXTENDEDTEXTMETRICS** escape. For more information about this structure, see the following Comments section.

Return Value

The return value specifies the number of **KERNTRACK** structures copied to the buffer. This value is zero if the font does not have kerning tracks defined or if the escape fails or is not implemented.

Comments

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

```
struct KERNTRACK {
    short Degree;
    short MinSize;
    short MinAmount;
    short MaxSize;
    short MaxAmount;
};
```

Following are the members in the **KERNTRACK** structure:

Degree
Specifies the amount of track kerning. Increasingly negative values represent tighter track kerning, and increasingly positive values represent looser track kerning.

MinSize
Specifies, in device units, the minimum font size for which linear track kerning applies.

MinAmount
Specifies, in font units, the amount of track kerning to apply to font sizes less than or equal to the size specified by the **MinSize** member.

MaxSize
Specifies, in device units, the maximum font size for which linear track kerning applies.

MaxAmount
Specifies, in font units, the amount of track kerning to apply to font sizes greater than or equal to the size specified by the **MaxSize** member.

Between the **MinSize** and **MaxSize** font sizes, track kerning is a linear function from **MinAmount** to **MaxAmount**. The values returned in the **KERNTRACK** structures are affected by whether relative character widths are enabled or disabled. For more information, see the description of the **ENABLERELATIVEWIDTHS** escape earlier in this chapter.

GETVECTORBRUSHSIZE

short Escape(*hdc***, GETVECTORBRUSHSIZE, sizeof(LOGBRUSH),** *lpInData***,** *lpOutData***)**

The **GETVECTORBRUSHSIZE** printer escape retrieves, in device units, the size of a plotter pen used to fill closed figures. Graphics device interface (GDI) uses this information to prevent the plotter pen from writing over the borders of the figure when filling closed figures.

Parameters

hdc
HDC Identifies the device context.

lpInData
LOGBRUSH FAR * Points to a **LOGBRUSH** structure that specifies the brush for which data is to be returned.

lpOutData
LPPOINT Points to a **POINT** structure whose **y** member contains the width of the pen, in device units.

Return Value

The return value specifies the outcome of the escape. This value is 1 if the escape is successful. It is zero if the escape is not successful or is not implemented.

GETVECTORPENSIZE

short Escape(*hdc*, GETVECTORPENSIZE, sizeof(LOGPEN), *lpInData*, *lpOutData*)

The **GETVECTORPENSIZE** printer escape retrieves the size, in device units, of a plotter pen. Graphics device interface (GDI) uses this information to prevent hatched brush patterns from overwriting the border of a closed figure.

Parameters

hdc
HDC Identifies the device context.

lpInData
LOGPEN FAR * Points to a **LOGPEN** structure that specifies the pen for which the width is to be retrieved.

lpOutData
LPPOINT Points to a **POINT** structure that contains in its second word the width of the pen, in device units.

Return Value

The return value specifies the outcome of the escape. This value is 1 if the escape is successful and zero if the escape is not successful or if it is not implemented.

MFCOMMENT

BOOL Escape(*hdc*, MFCOMMENT, *nCount*, *lpComment*, NULL)

The **MFCOMMENT** printer escape adds a comment to a metafile.

Parameters

hdc
HDC Identifies the device context for the device on which the metafile appears.

nCount
short Specifies the number of characters in the string pointed to by the *lpComment* parameter.

lpComment
LPSTR Points to a string that contains the comment that will appear in the metafile.

Return Value

The return value specifies the outcome of the escape. This value is –1 if an error, such as insufficient memory or an invalid port specification, occurs. Otherwise, it is positive.

MOUSETRAILS

short Escape(*hdc*, MOUSETRAILS, sizeof(WORD), *lpTrailSize*, NULL)

The **MOUSETRAILS** escape enables or disables mouse trails for display devices.

Parameters

hdc

HDC Identifies the device context.

lpTrailSize

LPINT points to a 16-bit variable containing a value specifying the action to take and the number of mouse cursor images to display (trail size). The variable can be one of the following values:

Value	Meaning
1 through 7	Enables mouse trails and sets the trail size to the specified number. A value of 1 requests a single mouse cursor. A value of 2 requests that one extra mouse cursor be drawn behind the current mouse cursor, and so on, up to a maximum of 7 total cursor images. The escape sets the MouseTrails entry in the WIN.INI file to the given value and returns the new trail size.
0	Disables mouse trails. The escape sets the MouseTrails entry to the negative value of the current trail size (if positive) and returns the negative value.
–1	Enables mouse trails. The display driver reads the MouseTrails entry from the [windows] section of the WIN.INI file. If the value of the entry is positive, the escape sets the trail size to the given value. If the entry is negative, the escape sets the trail size to the entry's absolute value and writes the positive value back to WIN.INI. If the MouseTrails entry is not found, the escape sets the trail size to 7 and writes a new MouseTrails entry to the WIN.INI file, setting its value to 7. The escape then returns the new trail size.
–2	Disables mouse trails but does not cause the display driver to update the WIN.INI file.
–3	Enables mouse trails but does not cause the display driver to update the WIN.INI file.

Return Value

The return value specifies the new trail size if the escape is successful. The return value is zero if the escape is not supported.

NEWFRAME

short Escape(*hdc*, NEWFRAME, NULL, NULL, NULL)

The **NEWFRAME** printer escape is maintained for backwards compatibility. Applications written for Windows 3.1 should use the **StartPage** and **EndPage** functions.

This escape informs the device that the application has finished writing to a page. It is typically used with a printer to direct the device driver to advance to a new page.

Parameters

hdc
HDC Identifies the device context.

Return Value

The return value specifies the outcome of the escape. This value is positive if the escape is successful. Otherwise, it is one of the following values:

Value	Meaning
SP_APPABORT	Job was terminated because the application's Abort function returned zero.
SP_ERROR	General error.
SP_OUTOFDISK	Not enough disk space is currently available for spooling, and no more space will become available.
SP_OUTOFMEMORY	Not enough memory is available for spooling.
SP_USERABORT	User terminated the job through Print Manager.

Comments

Do not use the **NEXTBAND** escape with the **NEWFRAME** escape. For banding device drivers, graphics device interface (GDI) replays a metafile to the printer, simulating a sequence of **NEXTBAND** escapes.

The **NEWFRAME** escape restores the default values of the device context. Consequently, if a font other than the default font is selected when the application calls the **NEWFRAME** escape, the application must select the font again following the **NEWFRAME** escape.

The **NEWFRAME** escape should not be used inside metafiles.

NEXTBAND

short Escape(*hdc*, NEXTBAND, NULL, NULL, *lpBandRect*)

The **NEXTBAND** printer escape informs the device driver that the application has finished writing to a band, causing the device driver to send the band to Print Manager and return the coordinates of the next band. Applications that process banding themselves use this escape.

Parameters

hdc

HDC Identifies the device context.

lpBandRect

LPRECT Points to the **RECT** structure that will receive the next band coordinates. The device driver copies the device coordinates of the next band into this structure.

Return Value

The return value specifies the outcome of the escape. This value is positive if the escape is successful. A return value of zero indicates that an error occurred. In addition, the following error values are defined:

Value	Meaning
SP_APPABORT	Job was terminated because the application's Abort function returned zero.
SP_ERROR	General error.
SP_OUTOFDISK	Not enough disk space is currently available for spooling, and no more space will become available.
SP_OUTOFMEMORY	Not enough memory is available for spooling.
SP_USERABORT	User terminated the job through Print Manager.

Comments

The **NEXTBAND** escape sets the band rectangle to the empty rectangle when printing reaches the end of a page.

Do not use the **NEWFRAME** escape with the **NEXTBAND** escape.

The **NEXTBAND** escape should not be used inside metafiles.

PASSTHROUGH

short Escape(*hdc*, PASSTHROUGH, NULL, *lpInData*, NULL)

The **PASSTHROUGH** printer escape allows the application to send data directly to the printer, bypassing the standard print-driver code.

Note To use this escape, an application must have complete information about how the particular printer operates.

Parameters

hdc

HDC Identifies the device context.

lpInData

LPSTR Points to a structure whose first word (16 bits) contains the number of bytes of input data. The remaining bytes of the structure contain the data itself.

Return Value

The return value specifies the number of bytes transferred to the printer if the escape is successful. This value is less than or equal to zero if the escape is not successful or not implemented.

Comments

There may be restrictions on the kinds of device data an application can send to the device without interfering with the operation of the driver. In general, applications must avoid resetting the printer or causing the page to be printed.

It is strongly recommended that applications do not perform actions that consume printer memory, such as downloading a font or a macro.

An application can avoid corrupting its data stream when issuing multiple, consecutive **PASSTHROUGH** escapes by not accessing the printer any other way during the sequence.

An application can guarantee that the **PASSTHROUGH** escape will be successful if it uses a "save" PostScript operator before sending **PASSTHROUGH** data and a "restore" operator after. Avoiding graphics device interface (GDI) functions between calls to the **PASSTHROUGH** escape and avoiding commands that cause a page to eject are other means to ensure that the escape will be successful.

POSTSCRIPT_DATA

The **POSTSCRIPT_DATA** printer escape is identical to the **PASSTHROUGH** escape.

POSTSCRIPT_IGNORE

short Escape(*hdc***, POSTSCRIPT_IGNORE, NULL,** *lpfOutput***, NULL)**

The **POSTSCRIPT_IGNORE** printer escape sets a flag indicating whether or not to suppress output.

Parameters

hdc
HDC Identifies the device context.

lpfOutput
BOOL FAR* Points to a flag indicating whether output should be suppressed. This value is nonzero to suppress output and zero otherwise.

Return Value

The return value specifies the previous setting of the output flag.

Comments

Applications that generate their own PostScript code can use the **POSTSCRIPT_IGNORE** escape to prevent the PostScript device driver from generating output.

QUERYESCSUPPORT

short Escape(*hdc***, QUERYESCSUPPORT, sizeof(int),** *lpEscNum***, NULL)**

The **QUERYESCSUPPORT** printer escape determines whether a particular escape is implemented by the device driver.

Parameters

hdc
HDC Identifies the device context.

lpEscNum
LPINT Points to a short integer that specifies the escape function to be checked.

Return Value

The return value specifies whether a particular escape is implemented. This value is nonzero for implemented escape functions. Otherwise, it is zero.

If the *lpEscNum* parameter is set to **DRAWPATTERNRECT**, the return value is one of the following values:

Value	Meaning
0	**DRAWPATTERNRECT** is not implemented.
1	**DRAWPATTERNRECT** is implemented for a printer other than the HP LaserJet IIP; this printer supports white rules.
2	**DRAWPATTERNRECT** is implemented for the HP LaserJet IIP.

RESTORE_CTM

short Escape(*hdc*, RESTORE_CTM, NULL, NULL, NULL)

The **RESTORE_CTM** printer escape restores the previously saved current transformation matrix.

The current transformation matrix controls the manner in which coordinates are translated, rotated, and scaled by the device. By using matrices, an application can combine these operations in any order to produce the desired mapping for a particular picture.

Parameters

hdc
HDC Identifies the device context.

Return Value

The return value specifies the number of **SAVE_CTM** escape calls without a corresponding **RESTORE_CTM** call. The return value is –1 if the escape is unsuccessful.

Comments

This escape is used only by PostScript printer drivers.

Applications should not make any assumptions about the initial contents of the current transformation matrix.

SAVE_CTM

short Escape(*hdc*, SAVE_CTM, NULL, NULL, NULL)

The **SAVE_CTM** printer escape saves the current transformation matrix.

The current transformation matrix controls the manner in which coordinates are translated, rotated, and scaled by the device. By using matrices, an application can combine these operations in any order to produce the desired mapping for a particular picture.

An application can restore the matrix by using the **RESTORE_CTM** escape.

An application typically saves the current transformation matrix before changing it. This allows the application to restore the previous state upon completion of a particular operation.

Parameters

hdc
HDC Identifies the device context.

Return Value

The return value specifies the number of **SAVE_CTM** escape calls without a corresponding **RESTORE_CTM** call. The return value is zero if the escape is unsuccessful.

Comments

This escape is used only by PostScript printer drivers.

Applications should not make any assumptions about the initial contents of the current transformation matrix.

Applications are expected to restore the contents of the current transformation matrix.

SELECTPAPERSOURCE

The **SELECTPAPERSOURCE** printer escape has been superseded by the **DeviceCapabilities** function (using the DC_BINS value). **SELECTPAPERSOURCE** is provided only for backward compatibility.

SETABORTPROC

short Escape(*hdc*, SETABORTPROC, NULL, *lpAbortFunc*, NULL)

The **SETABORTPROC** printer escape is maintained for backwards compatibility. Applications written for Windows 3.1 should use the **SetAbortProc** function.

This escape sets the Abort function for the print job.

To allow a print job to be canceled during spooling, an application must set the Abort function before the print job is started with the **STARTDOC** escape. Print Manager calls the Abort function during spooling to allow the application to cancel the print job or to take appropriate action for such errors as running out of disk space. If no Abort function is set, the print job will fail if there is not enough disk space for spooling.

Parameters

hdc
HDC Identifies the device context.

lpAbortFunc
FARPROC Points to the application-supplied Abort function. For details, see the following Comments section.

Return Value

The return value specifies the outcome of the escape. This value is greater than zero if the escape is successful. Otherwise, it is less than zero.

Comments

The address passed as the *lpAbortFunc* parameter must be created by using the **MakeProcInstance** function.

The callback function must use the Pascal calling convention and must be declared **FAR**. The Abort function must have the following form:

short FAR PASCAL *AbortFunc*(*hPr*,*code*)
HDC *hPr*;
short *code*;

AbortFunc is a placeholder for the application-supplied function name. The actual name must be exported by including it in an **EXPORTS** statement in the module-definition (.DEF) file for the application.

Following are the parameters in the Abort function:

hPr
Identifies the device context.

code
Specifies whether an error has occurred. This parameter is zero if no error has occurred. It is SP_OUTOFDISK if Print Manager is currently out of disk space and more disk space will become available if the application waits.

If *code* is SP_OUTOFDISK, the application does not have to abort the print job. If it does not abort the print job, it must yield to Print Manager by calling the **PeekMessage** or **GetMessage** function.

Return Value

The return value should be nonzero if the print job is to continue and zero if it is canceled.

SETALLJUSTVALUES

short Escape(*hdc***, SETALLJUSTVALUES, sizeof(EXTTEXTDATA),** *lpInData***, NULL)**

The **SETALLJUSTVALUES** printer escape is not recommended. Applications should use the **ExtTextOut** function instead of this escape. This escape sets all of the text-justification values that are used for text output in Windows 3.0 and earlier.

Text justification is the process of inserting extra pixels among break characters in a line of text. The space character is normally used as a break character.

Parameters

hdc
HDC Identifies the device context.

lpInData
EXTTEXTDATA FAR * Points to an **EXTTEXTDATA** structure that defines the text-justification values. For more information about this structure, see the Comments section.

Return Value

The return value specifies the outcome of the escape. This value is 1 if the escape is successful. Otherwise, it is zero.

Comments

The *lpInData* parameter points to an **EXTTEXTDATA** structure that describes the text-justification values used for text output. The **EXTTEXTDATA** structure has the following form:

```
typedef struct {
    short nSize;
    LPALLJUSTREC lpInData;
    LPFONTINFO lpFont;
    LPTEXTXFORM lpXForm;
    LPDRAWMODE lpDrawMode;
} EXTTEXTDATA;
```

This structure contains a **JUST_VALUE_STRUCT** structure that has the following form:

```
typedef struct {
    short nCharExtra;
    WORD cch;
    short nBreakExtra;
    WORD nBreakCount;
} JUST_VALUE_STRUCT;
```

Following are the members of **JUST_VALUE_STRUCT** structure:

nCharExtra
Specifies the total extra space, in font units, that must be distributed over **cch** characters.

cch
Specifies the number of characters over which the **nCharExtra** member is distributed.

nBreakExtra
Specifies the total extra space, in font units, that is distributed over **nBreakCount** characters.

nBreakCount
Specifies the number of break characters over which the **nBreakExtra** member is distributed.

The units used for the **nCharExtra** and **nBreakExtra** members are the font units of the device and are dependent on whether relative character widths were enabled with the **ENABLERELATIVEWIDTHS** escape.

The values set with this escape apply to subsequent calls to the **TextOut** function. The driver stops distributing the extra space specified in the **nCharExtra** member when it has output the number of characters specified in the **nCharCount**

member. Likewise, it stops distributing the space specified by the **nBreakExtra** member when it has output the number of characters specified by the **nBreakCount** member. A call on the same string to the **GetTextExtent** function made immediately after the call to the **TextOut** function will be processed in the same manner.

To reenable justification with the **SetTextJustification** and **SetTextCharacterExtra** functions, an application should call the **SETALLJUSTVALUES** escape and set the **nCharExtra** and **nBreakExtra** members to zero.

SET_ARC_DIRECTION

short Escape(*hdc***, SET_ARC_DIRECTION, sizeof(int),** *lpDirection***, NULL)**

The **SET_ARC_DIRECTION** printer escape specifies the direction in which elliptical arcs are drawn using the graphics device interface (GDI) **Arc** function.

By convention, elliptical arcs are drawn counterclockwise by GDI. This escape lets an application draw paths containing arcs drawn clockwise.

Parameters

hdc
HDC Identifies the device context.

lpDirection
LPINT Points to a short integer specifying the arc direction. It can be one of the following values:

COUNTERCLOCKWISE (0)
CLOCKWISE (1)

Return Value

The return value is the previous arc direction.

Comments

This escape maps to PostScript language elements and is intended for PostScript line devices.

SET_BACKGROUND_COLOR

short Escape(*hdc***, SET_BACKGROUND_COLOR,** *nCount***,** *lpNewColor***,** *lpOldColor***)**

The **SET_BACKGROUND_COLOR** printer escape sets and retrieves the current background color for the device.

The background color is the color of the screen surface before an application draws anything on the device. This escape is particularly useful for color printers and film recorders.

This escape should be sent before the application draws anything on the current page.

Parameters

hdc
HDC Identifies the device context.

nCount
int Specifies the number of bytes pointed to by the *lpNewColor* parameter.

lpNewColor
LPDWORD Points to a 32-bit integer specifying the desired background color. This parameter can be NULL if the application is merely retrieving the current background color.

lpOldColor
LPDWORD Points to a 32-bit integer that receives the previous background color. This parameter can be NULL if the application does not use the previous background color.

Return Value

The return value is nonzero if the escape is successful. This value is zero if it is unsuccessful.

Comments

The default background color is white.

The background color is reset to the default when the device driver receives an **ENDDOC** or **ABORTDOC** escape.

SET_BOUNDS

short Escape(*hdc*, SET_BOUNDS, sizeof(RECT), *lpInData*, NULL)

The **SET_BOUNDS** printer escape sets the bounding rectangle for the picture being produced by the device driver supporting the given device context. This escape is used when creating images in a file format such as Encapsulated PostScript (EPS) and Hewlett-Packard Graphics Language (HPGL) for which there is a device driver.

Parameters

hdc
HDC Identifies the device context.

lpInData
LPRECT Points to a **RECT** structure that specifies in device coordinates a rectangle that bounds the image to be output.

Return Value

The return value is nonzero if the escape was successful. Otherwise, it is zero.

Comments

An application should issue this escape before each page in the image. For single-page images, this escape should be issued immediately before the **STARTDOC** escape.

When an application uses coordinate-transformation escapes, device drivers may not perform bounding box calculations correctly. When an application uses the **SET_BOUNDS** escape, the driver does not have to calculate the bounding box.

Applications should always use this escape to ensure support for the Encapsulated PostScript (EPS) printing capabilities.

SET_CLIP_BOX

short Escape(*hdc*, SET_CLIP_BOX, sizeof(RECT), *lpClipBox*, (LPSTR) NULL)

The **SET_CLIP_BOX** printer escape sets the clipping rectangle or restores the previous clipping rectangle. This escape is implemented by printer drivers that use the coordinate-transformation escapes **TRANSFORM_CTM**, **SAVE_CTM**, and **RESTORE_CTM**.

When an application calls a graphics device interface (GDI) output function, GDI calculates a clipping rectangle bounding the primitive and passes both the primitive and the clipping rectangle to the printer driver. The printer driver is expected to clip the primitive to the specified bounding rectangle. However, when an application uses the coordinate-transformation escapes, the clipping rectangle calculated by GDI is usually invalid. An application can use the **SET_CLIP_BOX** escape to specify the correct clipping rectangle when coordinate transformations are used.

Parameters

hdc
HDC Identifies the device context.

lpClipBox
LPRECT Points to a **RECT** structure containing the bounding rectangle of the clipping region. If *lpClipBox* is not NULL, the previous clipping rectangle is saved and the current clipping rectangle is set to the specified bounds. If *lpClipBox* is NULL, the previous clipping rectangle is restored.

Return Value

The return value is nonzero if the clipping rectangle was properly set. Otherwise, it is zero.

Comments

This escape is used only by PostScript printer drivers.

SETCOLORTABLE

short Escape(*hdc***, SETCOLORTABLE, sizeof(COLORTABLE_STRUCT),** *lpInData***,** *lpColor***)**

The **SETCOLORTABLE** printer 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

hdc
HDC Identifies the device context.

lpInData
COLORTABLE_STRUCT FAR * Points to a structure that contains the index and RGB value of the color-table entry. For more information about the **COLORTABLE_STRUCT** structure, see the following Comments section.

lpColor
LPDWORD Points to the long integer that is to receive the RGB color value selected by the device driver to represent the requested color value.

Return Value

The return value specifies the outcome of the escape. This value is positive if the escape is successful. Otherwise, it is negative.

Comments

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

```
struct COLORTABLE_STRUCT {
    WORD  Index;
    DWORD rgb;
};
```

Following are the members of the **COLORTABLE_STRUCT** structure:

Index
Specifies the color-table index. Color-table entries start at zero for the first entry.

rgb
Specifies the desired RGB color value.

The color table for a device is a shared resource; changing the system display color for one window changes it for all windows. Only application developers who have a thorough knowledge of the display driver should use this escape.

The **SETCOLORTABLE** escape has no effect on devices with fixed color tables.

This escape is intended for use by both printer and display drivers. However, the EGA and VGA color drivers do not support it.

This escape changes the palette used by the display driver. However, because the color-mapping algorithms for the driver will probably no longer work with a different palette, an extension has been added to this function.

If the color index pointed to by the *lpInData* parameter is 0XFFFF, the driver is to leave all color-mapping functionality to the calling application. The application must use the proper color-mapping algorithm and take responsibility for passing the correctly mapped physical color to the driver (instead of the logical RGB color) in such device-driver functions as **RealizeObject** and **ColorInfo**.

For example, if the device supports 256 colors with palette indexes of 0 through 255, an application determines which index contains the color that it wants to use in a certain brush. It then passes this index in the low-order byte of the double-word logical color passed to the **RealizeObject** device-driver function. The driver uses this color exactly as passed instead of performing its usual color-mapping algorithm. If the application wants to reactivate the driver's color-mapping algorithm (that is, if it restores the original palette when switching from its window context), then the color index pointed to by *lpInData* should be 0xFFFE.

SETCOPYCOUNT

short Escape(*hdc***, SETCOPYCOUNT, sizeof(int),** *lpNumCopies***,** *lpActualCopies***)**

The **SETCOPYCOUNT** printer escape is maintained for backwards compatibility. Applications written for Windows 3.1 should use the **ExtDeviceMode** function.

This escape specifies the number of uncollated copies of each page that the printer is to print.

Parameters

hdc
HDC Identifies the device context.

lpNumCopies
LPINT Points to a short integer that contains the number of uncollated copies to be printed.

lpActualCopies
LPINT Points to a short integer that will receive the number of copies to be printed. This may be less than the number requested if the requested number is greater than the maximum copy count for the device.

Return Value

The return value specifies the outcome of the escape. It is 1 if the escape is successful and zero if the escape is not successful. The return value is zero if the escape is not implemented.

SETKERNTRACK

short Escape(*hdc***, SETKERNTRACK, sizeof(int),** *lpNewTrack***,** *lpOldTrack***)**

The **SETKERNTRACK** printer escape specifies which kerning track to use for drivers that support automatic track kerning. A kerning track of zero disables automatic track kerning.

When track kerning is enabled, the driver will automatically kern all characters according to the specified track. The driver will reflect this kerning both on the printer and in **GetTextExtent** function calls.

Parameters

hdc
HDC Identifies the device context.

lpNewTrack
LPINT Points to a short integer that specifies the kerning track to use. A value of zero disables this feature. Values in the range 1 through the value of the **etmKernTracks** member correspond to positions in the track-kerning table (using 1 as the first item in the table). For more information, see the description of the **EXTTEXTMETRIC** structure provided in the description of the **GETEXTENDEDTEXTMETRICS** escape earlier in this chapter.

lpOldTrack
LPINT Points to a short integer that will receive the previous kerning track.

Return Value

The return value specifies the outcome of the escape. It is 1 if the escape is successful and zero if the escape is not successful or not implemented.

Comments

Automatic track kerning is disabled by default.

A driver does not have to support the **SETKERNTRACK** escape just because it supplies the track-kerning table to the application by using the **GETTRACK-KERNTABLE** escape. In a case where **GETTRACKKERNTABLE** is supported but the **SETKERNTRACK** escape is not, the application must properly space the characters on the output device.

SETLINECAP

short Escape(*hdc***, SETLINECAP, sizeof(int),** *lpNewCap***,** *lpOldCap***)**

The **SETLINECAP** printer escape sets the line end cap.

A line end cap is that portion of a line segment that appears on either end of the segment. The cap may be square or circular. It can extend past or remain flush with the specified segment endpoints.

Parameters

hdc
HDC Identifies the device context.

lpNewCap
LPINT Points to a short integer that specifies the end-cap type. Following are the possible values and their meanings:

Value	Meaning
–1	Line segments are drawn by using the default graphics device interface (GDI) end cap.
0	Line segments are drawn with a squared end point that does not project past the specified segment length.
1	Line segments are drawn with a rounded end point; the diameter of this semicircular arc is equal to the line width.
2	Line segments are drawn with a squared end point that projects past the specified segment length. The projection is equal to half the line width.

lpOldCap
LPINT Points to a short integer that specifies the previous end-cap setting.

Return Value

The return value specifies the outcome of the escape. It is positive if the escape is successful. Otherwise, it is negative.

Comments

This escape is used only by PostScript printer drivers.

The interpretation of this escape varies with page-description languages (PDLs). For its exact meaning, consult the PDL documentation.

This escape is also known as **SETENDCAP**.

SETLINEJOIN

short Escape(*hdc***, SETLINEJOIN, sizeof(int),** *lpNewJoin***,** *lpOldJoin***)**

The **SETLINEJOIN** printer escape specifies how a device driver will join two intersecting line segments. The intersection can form a rounded, squared, or mitered corner.

Parameters

hdc
HDC Identifies the device context.

lpNewJoin
LPINT Points to a short integer that specifies the type of intersection. Following are the possible values and their meanings:

Value	Meaning
–1	Line segments are joined by using the default graphics device interface (GDI) setting.
0	Line segments are joined with a mitered corner; the outer edges of the lines extend until they meet at an angle. This is referred to as a miter join.
1	Line segments are joined with a rounded corner; a semicircular arc with a diameter equal to the line width is drawn around the point where the lines meet. This is referred to as a round join.
2	Line segments are joined with a squared end point; the outer edges of the lines are not extended. This is referred to as a bevel join.

lpOldJoin
LPINT Points to a short integer that specifies the previous line join setting.

Return Value

The return value specifies the outcome of the escape. It is positive if the escape is successful. Otherwise, it is negative.

Comments

This escape is used only by PostScript printer drivers.

The interpretation of this escape varies with page-description languages (PDLs). For its exact meaning, consult the PDL documentation.

If an application specifies a miter join but the angle of intersection is too small, the device driver ignores the miter setting and uses a bevel join instead.

SETMITERLIMIT

short Escape(*hdc***, SETMITERLIMIT, sizeof(int),** *lpNewMiter***,** *lpOldMiter***)**

The **SETMITERLIMIT** printer escape sets the miter limit for a device. The miter limit controls the angle at which a device driver replaces a miter join with a bevel join.

Parameters

hdc
HDC Identifies the device context.

lpNewMiter
LPINT Points to a short integer that specifies the desired miter limit. Only values greater than or equal to −1 are valid. If the value is −1, the driver will use the default graphics device interface (GDI) miter limit.

lpOldMiter
LPINT Points to a short integer that specifies the previous miter-limit setting.

Return Value

The return value specifies the outcome of the escape. This value is positive if the escape is successful. Otherwise, it is negative.

Comments

This escape is used only by PostScript printer drivers.

The miter limit is defined as follows:

miter length / line width = 1 / sin(x/2)

where x is the angle of the line join, in radians.

The interpretation of this escape varies with page-description languages (PDLs). For its exact meaning, consult the PDL documentation.

SET_POLY_MODE

short Escape(*hdc*, SET_POLY_MODE, sizeof(int), *lpMode*, NULL)

The **SET_POLY_MODE** printer escape sets the poly mode for the device driver. The poly mode is a state variable indicating how to interpret calls to graphics device interface (GDI) **Polygon** and **Polyline** functions.

The **SET_POLY_MODE** escape enables a device driver to draw shapes (such as Bezier curves) not directly supported by GDI. This permits applications that draw complex curves to send the curve description directly to a device without having to simulate the curve as a polygon with a large number of points.

Parameters

hdc
HDC Identifies the device context.

lpMode
LPINT Points to a short integer specifying the desired poly mode. The poly mode is a state variable indicating how points in **Polygon** or **Polyline** function calls should be interpreted. Device drivers are not required to support all possible modes. A device driver returns zero if it does not support the specified mode. The *lpMode* parameter may be one of the following values:

Value	Meaning
PM_POLYLINE (1)	Points define a conventional polygon or polyline.
PM_BEZIER (2)	Points define a sequence of 4-point Bezier spline curves. The first curve passes through the first four points, with the first and fourth points as endpoints and the second and third points as control points. Each subsequent curve in the sequence has the endpoint of the previous curve as its start point, the next two points as control points, and the third as its endpoint. The last curve in the sequence is permitted to have fewer than four points. If the curve has only one point, it is considered a point. If it has two points, it is a line segment. If it has three points, it is a parabola defined by drawing a Bezier curve with the first and third points as endpoints and the two control points equal to the second point.
PM_POLYLINESEGMENT (3)	Points specify a list of coordinate pairs. Line segments are drawn connecting each successive pair of points.
PM_POLYSCANLINE (4)	Points specify a list of coordinate pairs. Line segments are drawn connecting each successive pair of points. Each line segment is a nominal-width line drawn with the current brush. Each line segment must be strictly vertical or horizontal, and scan lines must be passed in strictly increasing or decreasing order. This mode is only used for polygon calls.

Return Value

The return value is the previous poly mode. If the return value is zero, the device driver did not handle the request.

Comments

This escape is used only by PostScript printer drivers.

An application should issue the **SET_POLY_MODE** escape before it draws a complex curve. It should then call the **Polyline** or **Polygon** function with the desired control points defining the curve. After drawing the curve, the application should reset the driver to its previous state by issuing the **SET_POLY_MODE** escape.

Polyline calls draw using the currently selected pen.

Polygon calls draw using the currently selected pen and brush. If the start point and endpoint are not equal, a line is drawn from the start point to the endpoint before the polygon (or curve) is filled.

GDI treats **Polygon** calls using PM_POLYLINESEGMENT mode exactly the same as **Polyline** calls.

Four points define a Bezier curve. GDI generates the curve by connecting the first and second, second and third, and third and fourth points. GDI then connects the midpoints of these consecutive line segments. Finally, GDI connects the midpoints of the lines connecting the midpoints, and so forth.

The line segments drawn in this way converge to a curve defined by the following parametric equations, expressed as a function of the independent variable *t*.

$$X(t) = (1-t)^3 x_1 + 3(1-t)^2 t x_2 + 3(1-t)t^2 x_3 + t^3 x_4$$

$$Y(t) = (1-t)^3 y_1 + 3(1-t)^2 t y_2 + 3(1-t)t^2 y_3 + t^3 y_4$$

The points (x_1, y_1), (x_2, y_2), (x_3, y_3) and (x_4, y_4) are the control points defining the curve. The independent variable t varies from 0 to 1.

Primitive types other than PM_BEZIER and PM_POLYLINESEGMENT may be added to this escape in the future. Applications should check the return value from this escape to determine whether the driver supports the specified poly mode.

SET_SCREEN_ANGLE

short Escape(*hdc*, SET_SCREEN_ANGLE, sizeof(int), *lpAngle*, NULL)

The **SET_SCREEN_ANGLE** printer escape sets the current screen angle to the desired angle and enables an application to simulate the tilting of a photographic mask in producing a color separation for a particular primary.

Parameters

hdc
HDC Identifies the device context.

lpAngle
LPINT Points to a short integer specifying the desired screen angle in tenths of a degree. The angle is measured counterclockwise.

Return Value

The return value is the previous screen angle.

Comments

Four-color process separation is the process of separating the colors comprising an image into four component primaries: cyan, magenta, yellow, and black. The image is then reproduced by overprinting each primary.

In traditional four-color process printing, half-tone images for each of the four primaries are photographed against a mask tilted to a particular angle. Tilting the mask in this manner minimizes unwanted moiré patterns caused by overprinting two or more colors.

The device driver defines the default screen angle.

SET_SPREAD

short Escape(*hdc*, SET_SPREAD, sizeof(int), *lpSpread*, NULL)

The **SET_SPREAD** printer escape sets the amount that nonwhite primitives are expanded for a given device to provide a slight overlap between primitives to compensate for imperfections in the reproduction process.

Spot color separation is the process of separating an image into each distinct color used in the image. The image is reproduced by overprinting each successive color in the image.

When reproducing a spot-separated image, the printing equipment must be calibrated to align each page exactly on each pass. However, differences in temperature, humidity, and so forth between passes often cause images to align imperfectly on subsequent passes. For this reason, lines in spot separations are often widened (spread) slightly to make up for problems in registering subsequent passes through the printer. This process is called trapping. The **SET_SPREAD** escape implements this process.

Parameters

hdc
HDC Identifies the device context.

lpSpread
LPINT Points to a short integer that specifies the amount, in pixels, by which all nonwhite primitives are to be expanded.

Return Value

The return value is the previous spread value.

Comments

The default spread is zero.

The current spread applies to all bordered primitives (whether or not the border is visible) and text.

STARTDOC

short Escape(*hdc***, STARTDOC,** *nCount***,** *lpDocName***, NULL)**

The **STARTDOC** printer escape is maintained for backwards compatibility. Applications written for Windows 3.1 should use the **StartDoc** function.

This escape informs the device driver that a new print job is starting and that all subsequent **NEWFRAME** escape calls should be spooled under the same job until an **ENDDOC** escape call occurs. This ensures that documents longer than one page will not be interspersed with other jobs.

Parameters

hdc
HDC Identifies the device context.

nCount
short Specifies the number of characters in the string pointed to by the *lpDocName* parameter.

lpDocName
LPSTR Points to a null-terminated string that specifies the name of the document. The document name is displayed in the Print Manager window. The maximum length of this string is 31 characters plus the terminating null character.

Return Value

The return value specifies the outcome of the escape. It is –1 if an error such as insufficient memory or an invalid port specification occurs. Otherwise, it is positive.

Comments

Following is the correct sequence of events in a printing operation:

1. Create the device context.
2. Set the Abort function to keep out-of-disk-space errors from terminating a printing operation.

   An Abort procedure that handles these errors must be set by using the **SETABORTPROC** escape.
3. Begin the printing operation with the **STARTDOC** escape.
4. Begin each new page with the **NEWFRAME** escape or each new band with the **NEXTBAND** escape.
5. End the printing operation with the **ENDDOC** escape.
6. Destroy the Cancel dialog box, if any.
7. Free the procedure-instance address of the Abort function.

If an application encounters a printing error or a canceled print operation, it must not attempt to terminate the operation by using the **Escape** function with either the **ENDDOC** or **ABORTDOC** escape. Graphics device interface (GDI) automatically terminates the operation before returning the error value.

The **STARTDOC** escape should not be used inside metafiles.

STRETCHBLT

The **STRETCHBLT** printer escape is provided for backwards compatibility. Applications should use the **StretchBlt** function instead of this escape.

See Also **StretchBlt**

TRANSFORM_CTM

short Escape(*hdc***, TRANSFORM_CTM, 36,** *lpMatrix***, NULL)**

The **TRANSFORM_CTM** printer escape modifies the current transformation matrix. The current transformation matrix controls the manner in which coordinates are translated, rotated, and scaled by the device. By using matrices, you can combine these operations in any order to produce the desired mapping for a particular picture.

The new current transformation matrix will contain the product of the matrix referenced by the *lpMatrix* parameter and the previous current transformation matrix (CTM = M * CTM).

Parameters

hdc

HDC Identifies the device context.

lpMatrix

LPSTR Points to a 3-by-3 array of 32-bit integer values specifying the new transformation matrix. Entries in the matrix are scaled to represent fixed-point real numbers. Each matrix entry is scaled by 65,536. The high-order word of the entry contains the whole integer portion, and the low-order word contains the fractional portion.

Return Value

The return value is nonzero if the escape was successful and zero if it was unsuccessful.

Comments

This escape is used only by PostScript printer drivers.

When an application modifies the current transformation matrix, it must specify the clipping rectangle by issuing the **SET_CLIP_BOX** escape.

Applications should not make any assumptions about the initial value of the current transformation matrix.

Dynamic Data Exchange Transactions

Chapter 6

The Dynamic Data Exchange Management Library (DDEML) notifies an application of dynamic data exchange (DDE) activity that affects the application by sending transactions to the application's DDE callback function. A transaction is similar to a message—it is a named constant accompanied by other parameters that contain additional information about the transaction.

This chapter lists the DDE transactions in alphabetic order.

XTYP_ADVDATA

3.1

```
#include <ddeml.h>

XTYP_ADVDATA
hszTopic = hsz1;      /* handle of topic-name string */
hszItem = hsz2;       /* handle of item-name string  */
hDataAdvise = hData; /* handle of the advise data   */
```

A client's DDE callback function can receive this transaction after the client has established an advise loop with a server. This transaction informs the client that the value of the data item has changed.

Parameters

hszTopic
Value of *hsz1*. Identifies the topic name.

hszItem
Value of *hsz2*. Identifies the item name.

hDataAdvise
Value of *hData*. Identifies the data associated with the topic/item name pair. If the client specified the XTYPF_NODATA flag when it requested the advise loop, this parameter is NULL.

Return Value

A DDE callback function should return DDE_FACK if it processes this transaction, DDE_FBUSY if it is too busy to process this transaction, or DDE_FNOTPROCESSED if it denies this transaction.

Comments

An application need not free the data handle obtained during this transaction. If the application needs to process the data after the callback function returns, however, it must copy the data associated with the data handle. An application can use the **DdeGetData** function to copy the data.

See Also

DdeClientTransaction, DdePostAdvise

XTYP_ADVREQ

3.1

```
#include <ddeml.h>

XTYP_ADVREQ
hszTopic = hsz1;           /* handle of topic-name string     */
hszItem = hsz2;            /* handle of item-name string      */
cAdvReq = LOWORD(dwData1); /* count of remaining transactions */
```

The system sends this transaction to a server after the server calls the **DdePostAdvise** function. This transaction informs the server that an advise transaction is outstanding on the specified topic/item name pair and that data corresponding to the topic/item name pair has changed.

Parameters

hszTopic
Value of *hsz1*. Identifies the topic name.

hszItem
Value of *hsz2*. Identifies the item name that has changed.

cAdvReq
Value of the low-order word of *dwData1*. Specifies the count of XTYP_ADVREQ transactions that remain to be processed on the same topic/item/format name set, within the context of the current call to the **DdePostAdvise** function. If the current XTYP_ADVREQ transaction is the last one, the count is zero. A server can use this count to determine whether to create an HDATA_APPOWNED data handle for the advise data.

If the DDEML issued the XTYP_ADVREQ transaction because of a late-arriving DDE_FACK transaction flag from a client, the low-order word is set to CADV_LATEACK. The DDE_FACK transaction flag arrives late when a server is sending information faster than a client can process it.

Return Value

The server should call the **DdeCreateDataHandle** function to create a data handle that identifies the changed data and then should return the handle. If the server is unable to complete the transaction, it should return NULL.

Comments

A server cannot block this transaction type; the CBR_BLOCK return value is ignored.

See Also

DdeCreateDataHandle, **DdeInitialize**, **DdePostAdvise**

XTYP_ADVSTART

3.1

```
#include <ddeml.h>

XTYP_ADVSTART
hszTopic = hsz1;      /* handle of topic-name string */
hszItem = hsz2;       /* handle of item-name string  */
```

A server's DDE callback function receives this transaction when a client specifies XTYP_ADVSTART for the *wType* parameter of the **DdeClientTransaction** function. A client uses this transaction to establish an advise loop with a server.

Parameters

hszTopic
Value of *hsz1*. Identifies the topic name.

hszItem
Value of *hsz2*. Identifies the item name.

Return Value

To allow an advise loop on the specified topic/item name pair, a server's DDE callback function should return a nonzero value. To deny the advise loop, it should return zero. If the callback function returns a nonzero value, any subsequent call by the server to the **DdePostAdvise** function on the same topic/item name pair will cause the system to send a XTYP_ADVREQ transaction to the server.

Comments

If a client requests an advise loop on a topic/item/format name set for which an advise loop is already established, the DDEML does not create a duplicate advise loop. Instead, the DDEML alters the advise loop flags (XTYPF_ACKREQ and XTYPF_NODATA) to match the latest request.

If the server application specified the CBF_FAIL_ADVISES flag in the **DdeInitialize** function, this transaction is filtered.

See Also

DdeClientTransaction, **DdeInitialize**, **DdePostAdvise**

XTYP_ADVSTOP

3.1

```
#include <ddeml.h>

XTYP_ADVSTOP
hszTopic = hsz1;      /* handle of topic-name string */
hszItem = hsz2;       /* handle of item-name string  */
```

A server's DDE callback function receives this transaction when a client specifies XTYP_ADVSTOP for the *wType* parameter of the **DdeClientTransaction** function. A client uses this transaction to end an advise loop with a server.

Parameters

hszTopic
Value of *hsz1*. Identifies the topic name.

hszItem
Value of *hsz2*. Identifies the item name.

Return Value

This transaction does not return a value.

Comments

If the server application specified the CBF_FAIL_ADVISES flag in the **DdeInitialize** function, this transaction is filtered.

See Also

DdeClientTransaction, DdeInitialize, DdePostAdvise

XTYP_CONNECT

3.1

```
#include <ddeml.h>

XTYP_CONNECT
hszTopic = hsz1;                  /* handle of topic-name string      */
hszService = hsz2;                /* handle of service-name string    */
pcc = (CONVCONTEXT FAR *)dwData1; /* address of CONVCONTEXT structure */
fSameInst = (BOOL) dwData2;       /* same instance flag               */
```

A server's DDE callback function receives this transaction when a client specifies a service name that the server supports and a topic name that is not set to NULL in a call to the **DdeConnect** function.

Parameters

hszTopic
Value of *hsz1*. Identifies the topic name.

hszService
Value of *hsz2*. Identifies the service name.

pcc
Value of *dwData1*. Points to a **CONVCONTEXT** data structure that contains context information for the conversation. If the client is not a DDEML application, this parameter should be set to zero.

fSameInst
Value of *dwData2*. Specifies whether the client is the same application instance as the server. If this parameter is TRUE, the client is the same instance; if this parameter is FALSE, the client is a different instance.

Return Value

To allow the client to establish a conversation on the specified service/topic name pair, a server's DDE callback function should return a nonzero value. To deny the conversation, it should return zero. If the callback function returns a nonzero value and a conversation is successfully established, the system passes the conversation handle to the server by issuing an XTYP_CONNECT_CONFIRM transaction to the server's DDE callback function (unless the server specified the CBF_FAIL_CONNECT_CONFIRMS flag in the **DdeInitialize** function).

Comments

If the server application specified the CBF_FAIL_CONNECTIONS flag in the **DdeInitialize** function, this transaction is filtered.

A server cannot block this transaction type; the CBR_BLOCK return value is ignored.

See Also

DdeConnect, **DdeInitialize**

XTYP_CONNECT_CONFIRM

3.1

```
#include <ddeml.h>

XTYP_CONNECT_CONFIRM
hszTopic = hsz1;            /* handle of topic-name string   */
hszService = hsz2;          /* handle of service-name string */
fSameInst = (BOOL) dwData2; /* same instance flag            */
```

A server's DDE callback function receives this transaction to confirm that a conversation has been established with a client and to provide the server with the conversation handle. The system sends this transaction as a result of a previous XTYP_CONNECT or XTYP_WILDCONNECT transaction.

Parameters

hszTopic
Value of *hsz1*. Identifies the topic name on which the conversation has been established.

hszService
Value of *hsz2*. Identifies the service name on which the conversation has been established.

fSameInst
Value of *dwData2*. Specifies whether the client is the same application instance as the server. If this parameter is a nonzero value, the client is the same instance. If this parameter is zero, the client is a different instance.

Return Value

This transaction does not return a value.

Comments

If the server application specified the CBF_FAIL_CONFIRMS flag in the **DdeInitialize** function, this transaction is filtered.

A server cannot block this transaction type; the CBR_BLOCK return value is ignored.

See Also

DdeConnect, **DdeConnectList**, **DdeInitialize**

XTYP_DISCONNECT

3.1

```
#include <ddeml.h>

XTYP_DISCONNECT
fSameInst = (BOOL) dwData2;  /* same instance flag */
```

An application's DDE callback function receives this transaction when the application's partner in a conversation uses the **DdeDisconnect** function to terminate the conversation.

Parameters

fSameInst
Value of *dwData2*. Specifies whether the partners in the conversation are the same application instance. If this parameter is TRUE, the partners are the same instance. If this parameter is FALSE, the partners are different instances.

Return Value

This transaction does not return a value.

Comments

If the application specified the CBF_SKIP_DISCONNECTS flag in the **DdeInitialize** function, this transaction is filtered.

The application can obtain the status of the terminated conversation by calling the **DdeQueryConvInfo** function while processing this transaction. The conversation handle becomes invalid after the callback function returns.

An application cannot block this transaction type; the CBR_BLOCK return value is ignored.

See Also **DdeDisconnect**, **DdeQueryConvInfo**

XTYP_ERROR

3.1

```
#include <ddeml.h>

XTYP_ERROR
wErr = LOWORD(dwData1); /* error value */
```

A DDE callback function receives this transaction when a critical error occurs.

Parameters

wErr
Value of *dwData1*. Specifies the error value. Currently, only the DMLERR_LOW_MEMORY error value is supported. It means that memory is low—advise, poke, or execute data may be lost, or the system may fail.

Return Value This transaction does not return a value.

Comments An application cannot block this transaction type; the CBR_BLOCK return value is ignored. The DDEML attempts to free memory by removing noncritical resources. An application that has blocked conversations should unblock them.

XTYP_EXECUTE

3.1

```
#include <ddeml.h>

XTYP_EXECUTE
hszTopic = hsz1;    /* handle of the topic-name string */
hDataCmd = hData;   /* handle of the command string    */
```

A server's DDE callback function receives this transaction when a client specifies XTYP_EXECUTE for the *wType* parameter of the **DdeClientTransaction** function. A client uses this transaction to send a command string to the server.

Parameters

hszTopic
Value of *hsz1*. Identifies the topic name.

hDataCmd
Value of *hData*. Identifies the command string.

Return Value

A server's DDE callback function should return DDE_FACK if it processes this transaction, DDE_FBUSY if it is too busy to process this transaction, or DDE_FNOTPROCESSED if it denies this transaction.

Comments

If the server application specified the CBF_FAIL_EXECUTES flag in the **DdeInitialize** function, this transaction is filtered.

An application need not free the data handle obtained during this transaction. If the application needs to process the string after the callback function returns, however, the application must copy the command string associated with the data handle. An application can use the **DdeGetData** function to copy the data.

See Also

DdeClientTransaction, **DdeInitialize**

XTYP_MONITOR

3.1

```
#include <ddeml.h>

XTYP_MONITOR
hDataEvent = hData;     /* handle of event data */
fwEvent = dwData2;      /* event flag           */
```

The DDE callback function of a DDE debugging application receives this transaction whenever a DDE event occurs in the system. An application can receive this transaction only if it specified the APPCLASS_MONITOR flag when it called the **DdeInitialize** function.

Parameters

hDataEvent
Value of *hData*. Identifies a global memory object that contains information about the DDE event. The application should use the **DdeAccessData** function to obtain a pointer to the object.

fwEvent
Value of *dwData2*. Specifies the DDE event. This parameter may be one of the following values:

Value	Meaning
MF_CALLBACKS	The system sent a transaction to a DDE callback function. The global memory object contains a **MONCBSTRUCT** structure that provides information about the transaction.
MF_CONV	A DDE conversation was established or terminated. The global memory object contains a **MONCONVSTRUCT** structure that provides information about the conversation.
MF_ERRORS	A DDE error occurred. The global memory object contains a **MONERRSTRUCT** structure that provides information about the error.
MF_HSZ_INFO	A DDE application created or freed a string handle or incremented the use count of a string handle, or a string handle was freed as a result of a call to the **DdeUninitialize** function. The global memory object contains a **MONHSZSTRUCT** structure that provides information about the string handle.
MF_LINKS	A DDE application started or ended an advise loop. The global memory object contains a **MONLINKSTRUCT** structure that provides information about the advise loop.
MF_POSTMSGS	The system or an application posted a DDE message. The global memory object contains a **MONMSGSTRUCT** structure that provides information about the message.
MF_SENDMSGS	The system or an application sent a DDE message. The global memory object contains a **MONMSGSTRUCT** structure that provides information about the message.

Return Value The callback function should return zero if it processes this transaction.

See Also **DdeAccessData**, **DdeInitialize**

XTYP_POKE

3.1

```
#include <ddeml.h>

XTYP_POKE
hszTopic = hsz1;      /* handle of topic-name string */
hszItem = hsz2;       /* handle of item-name string  */
hDataPoke = hData;    /* handle of data for server   */
```

A server's DDE callback function receives this transaction when a client specifies XTYP_POKE as the *wType* parameter of the **DdeClientTransaction** function. A client uses this transaction to send unsolicited data to the server.

Parameters

hszTopic
Value of *hsz1*. Identifies the topic name.

hszItem
Value of *hsz2*. Identifies the item name.

hDataPoke
Value of *hData*. Identifies the data that the client is sending to the server.

Return Value

A server's DDE callback function should return DDE_FACK if it processes this transaction, DDE_FBUSY if it is too busy to process this transaction, or DDE_FNOTPROCESSED if it denies this transaction.

Comments

If the server application specified the CBF_FAIL_POKES flag in the **DdeInitialize** function, this transaction is filtered.

See Also

DdeClientTransaction, DdeInitialize

XTYP_REGISTER

3.1

```
#include <ddeml.h>

XTYP_REGISTER
hszBaseServName = hsz1; /* handle of base service-name string     */
hszInstServName = hsz2; /* handle of instance service-name string */
```

A DDE callback function receives this transaction type whenever a DDEML server application uses the **DdeNameService** function to register a service name or whenever a non-DDEML application that supports the System topic is started.

Parameters

hszBaseServName
Value of *hsz1*. Identifies the base service name being registered.

hszInstServName
Value of *hsz2*. Identifies the instance-specific service name being registered.

Return Value

This transaction does not return a value.

Comments

If the application specified the CBF_SKIP_REGISTRATIONS flag in the **DdeInitialize** function, this transaction is filtered.

An application cannot block this transaction type; the CBR_BLOCK return value is ignored.

An application should use the *hszBaseServName* parameter to add the service name to the list of servers available to the user. An application should use the *hszInstServName* parameter to identify which application instance has started.

See Also **DdeInitialize**, **DdeNameService**

XTYP_REQUEST

3.1

```
#include <ddeml.h>

XTYP_REQUEST
hszTopic = hsz1;      /* handle of topic-name string */
hszItem = hsz2;       /* handle of item-name string  */
```

A DDE server callback function receives this transaction when a client specifies XTYP_REQUEST for the *wType* parameter of the **DdeClientTransaction** function. A client uses this transaction to request data from a server.

Parameters

hszTopic
Value of *hsz1*. Identifies the topic name.

hszItem
Value of *hsz2*. Identifies the item name that has changed.

Return Value The server should call the **DdeCreateDataHandle** function to create a data handle that identifies the changed data and then should return the handle. The server should return NULL if it is unable to complete the transaction. If the server returns NULL, the client receives a DDE_FNOTPROCESSED acknowledgment flag.

Comments If the server application specified the CBF_FAIL_REQUESTS flag in the **DdeInitialize** function, this transaction is filtered.

If responding to this transaction requires lengthy processing, the server can return CBR_BLOCK to suspend future transactions on the current conversation and then process the transaction asynchronously. When the server has finished and the data is ready to pass to the client, the server can call the **DdeEnableCallback** function to resume the conversation.

See Also **DdeClientTransaction**, **DdeCreateDataHandle**, **DdeEnableCallback**, **DdeInitialize**

XTYP_UNREGISTER

3.1

```
#include <ddeml.h>

XTYP_UNREGISTER
hszBaseServName = hsz1; /* handle of base service-name string     */
hszInstServName = hsz2; /* handle of instance service-name string */
```

A DDE callback function receives this transaction type whenever a DDEML server application uses the **DdeNameService** function to unregister a service name or whenever a non-DDEML application that supports the System topic is terminated.

Parameters

hszBaseServName
Value of *hsz1*. Identifies the base service name being unregistered.

hszInstServName
Value of *hsz2*. Identifies the instance-specific service name being unregistered.

Return Value

This transaction does not return a value.

Comments

If the application specified the CBF_SKIP_REGISTRATIONS flag in the **DdeInitialize** function, this transaction is filtered.

An application cannot block this transaction type; the CBR_BLOCK return value is ignored.

An application should use the *hszBaseServName* parameter to remove the service name from the list of servers available to the user. An application should use the *hszInstServName* parameter to identify which application instance has terminated.

See Also

DdeInitialize, **DdeNameService**

XTYP_WILDCONNECT

3.1

```
#include <ddeml.h>

XTYP_WILDCONNECT
hszTopic = hsz1;                  /* handle of topic-name string      */
hszService = hsz2;                /* handle of service-name string    */
pcc = (CONVCONTEXT FAR *)dwData1; /* address of CONVCONTEXT structure */
fSameInst = (BOOL) dwData2;       /* same-instance flag               */
```

A server's DDE callback function receives this transaction when a client specifies a service name that is set to NULL, a topic name that is set to NULL, or both in a call to the **DdeConnect** function. This transaction allows a client to establish a conversation on each of the server's service/topic name pairs that matches the specified service name and topic name.

Parameters

hszTopic
Value of *hsz1*. Identifies the topic name. If this parameter is NULL, the client is requesting a conversation on all topic names that the server supports.

hszService
Value of *hsz2*. Identifies the service name. If this parameter is NULL, the client is requesting a conversation on all service names that the server supports.

pcc
Value of *dwData1*. Points to a **CONVCONTEXT** data structure that contains context information for the conversation. If the client is not a DDEML application, this parameter is set to zero.

fSameInst
Value of *dwData2*. Specifies whether the client is the same application instance as the server. If this parameter is TRUE, the client is same instance. If this parameter is FALSE, the client is a different instance.

Return Value

The server should return a data handle that identifies an array of **HSZPAIR** structures. The array should contain one structure for each service/topic name pair that matches the service/topic name pair requested by the client. The array must be terminated by a NULL string handle. The system sends the XTYP_CONNECT_CONFIRM transaction to the server to confirm each conversation and to pass the conversation handles to the server. If the server specified the CBF_SKIP_CONNECT_CONFIRMS flag in the **DdeInitialize** function, it cannot receive these confirmations.

To refuse the XTYP_WILDCONNECT transaction, the server should return NULL.

Comments

If the server application specified the CBF_FAIL_CONNECTIONS flag in the **DdeInitialize** function, this transaction is filtered.

A server cannot block this transaction type; the CBR_BLOCK return code is ignored.

See Also

DdeConnect, DdeInitialize

XTYP_XACT_COMPLETE

3.1

```
#include <ddeml.h>

XTYP_XACT_COMPLETE
hszTopic = hsz1;        /* handle of topic-name string */
hszItem = hsz2;         /* handle of item-name string  */
hDataXact = hData;      /* handle of transaction data  */
dwXactID = dwData1;     /* transaction identifier      */
fwStatus = dwData2;     /* status flag                 */
```

A DDE client callback function receives this transaction when an asynchronous transaction, initiated by a call to the **DdeClientTransaction** function, has concluded.

Parameters

hszTopic
Value of *hsz1*. Identifies the topic name involved in the completed transaction.

hszItem
Value of *hsz2*. Identifies the item name involved in the completed transaction.

hDataXact
Value of *hData*. Identifies the data involved in the completed transaction, if applicable. If the transaction was successful but involved no data, this parameter is TRUE. If the transaction was unsuccessful, this parameter is NULL.

dwXactID
Value of *dwData1*. Contains the transaction identifier of the completed transaction.

fwStatus
Value of *dwData2*. Contains any applicable DDE_ status flags in the low-order word. This provides support for applications dependent on DDE_APPSTATUS bits. It is recommended that applications no longer use these bits—future versions of the DDEML may not support them.

Return Value

This transaction does not return a value.

Comments

An application need not free the data handle obtained during this transaction. If the application needs to process the data after the callback function returns, however, the application must copy the data associated with the data handle. An application can use the **DdeGetData** function to copy the data.

See Also

DdeClientTransaction

File Manager Events and Messages

Chapter 7

File Manager communicates with a File Manager extension dynamic-link library (DLL) by sending events and menu commands to the DLL's **FMExtensionProc** function. While processing an event or command, the DLL can retrieve information from File Manager by sending File Manager messages using the **SendMessage** function. This chapter provides complete descriptions of both the events and messages for File Manager in Microsoft Windows operating system, version 3.1

7.1 File Manager Events

This section lists File Manager events in alphabetic order.

FMEVENT_INITMENU

The FMEVENT_INITMENU message is sent to an extension dynamic-link library (DLL) when the user selects the menu for the extension from File Manager's menu bar. The extension can use this notification to initialize menu items in the menu.

Parameters

lParam
Specifies the menu handle in the high-order word. The low-order word specifies the delta value for the menu item.

Return Value

This message does not return a value.

Comments

An extension receives this message only when the user selects the top-level menu. If the extension contains submenus, it must initialize them at the same time as the top-level menu.

See Also

FMExtensionProc

FMEVENT_LOAD

The FMEVENT_LOAD message is sent to an extension dynamic-link library (DLL) when File Manager is loading the DLL.

Parameters

lParam
Points to an **FMS_LOAD** structure that specifies the menu-item delta value. An extension DLL should save the menu-item delta value and fill the other structure members with information about the extension. The **FMS_LOAD** structure has the following form:

```
#include <wfext.h>

typedef struct tagFMS_LOAD { /* fmsld */
    DWORD dwSize;
    char  szMenuName[MENU_TEXT_LEN];
    HMENU hMenu;
    UINT  wMenuDelta;
} FMS_LOAD;
```

Return Value

This message does not return a value.

Comments

An application should fill the **dwSize**, **szMenuName**, and **hMenu** members. It should also save the value of the **wMenuDelta** member and use it to identify menu items when modifying the menu. For more information, see the description of the **FMS_LOAD** structure.

See Also

FMExtensionProc

FMEVENT_SELCHANGE

The FMEVENT_SELCHANGE message is sent to an extension dynamic-link library (DLL) when the user selects a filename in File Manager's directory window or Search Results window.

Parameters

lParam
Not used.

Return Value

This message does not return a value.

Comments Changes in the tree half of the directory window do not produce this message.

Because the user can change the selection many times, the extension DLL must return promptly after processing this message to avoid slowing the selection process for the user.

See Also **FMExtensionProc**

FMEVENT_UNLOAD

The FMEVENT_UNLOAD message is sent to an extension dynamic-link library (DLL) when File Manager is unloading the DLL.

Parameters *lParam*
Not used.

Return Value This message does not return a value.

Comments The *hwnd* and *hMenu* values passed with the FMEVENT_LOAD and FMEVENT_INITMENU messages may not be valid at the time of this message.

See Also **FMExtensionProc**

FMEVENT_USER_REFRESH

The FMEVENT_USER_REFRESH message is sent to an extension dynamic-link library (DLL) when the user invokes File Manager's Refresh command in the Window menu. The extension can use this notification to update its menu.

Parameters *lParam*
Not used.

Return Value This message does not return a value.

See Also **FMExtensionProc**

7.2 File Manager Messages

This section lists File Manager messages in alphabetic order.

FM_GETDRIVEINFO

A File Manager extension sends an FM_GETDRIVEINFO message to retrieve drive information from the active File Manager window.

Parameters

wParam
Not used.

lParam
Points to an **FMS_GETDRIVEINFO** structure that receives drive information. The **FMS_GETDRIVEINFO** structure has the following form:

```
#include <wfext.h>

typedef struct tagFMS_GETDRIVEINFO { /* fmsgdi */
    DWORD dwTotalSpace;
    DWORD dwFreeSpace;
    char  szPath[260];
    char  szVolume[14];
    char  szShare[128];
} FMS_GETDRIVEINFO, FAR *LPFMS_GETDRIVEINFO;
```

Return Value

The return value is always nonzero.

See Also

FMExtensionProc

FM_GETFILESEL

A File Manager extension sends an FM_GETFILESEL message to retrieve information about a selected file from the active File Manager window (either the directory window or the Search Results window).

Parameters

wParam
Specifies the zero-based index of the selected file to retrieve.

lParam
Points to an **FMS_GETFILESEL** structure that receives information about the selection. The **FMS_GETFILESEL** structure has the following form:

```
#include <wfext.h>

typedef struct tagFMS_GETFILESEL { /* fmsgfs */
    UINT  wTime;
    UINT  wDate;
    DWORD dwSize;
    BYTE  bAttr;
    char  szName[260];
} FMS_GETFILESEL;
```

Return Value

The return value is the zero-based index of the selected file that was retrieved.

Comments

An extension can use the FM_GETSELCOUNT message to obtain the count of selected files.

The **szName** member of the **FMS_GETFILESEL** structure consists of an OEM character string. Before displaying this string, an extension should use the **OemToAnsi** function to convert the string to a Windows ANSI character string. If a string is to be passed to the file system (MS-DOS), an extension should not convert it.

See Also

FMExtensionProc, FM_GETFILESELLFN, FM_GETSELCOUNT, FM_GETSELCOUNTLFN, **OemToAnsi**

FM_GETFILESELLFN

A File Manager extension sends an FM_GETFILESELLFN message to retrieve information about a selected file from the active File Manager window (either the directory window or the Search Results window). The selected file can have a long filename.

Parameters

wParam
Specifies the zero-based index of the selected file to retrieve.

lParam
Points to an **FMS_GETFILESEL** structure that receives information about the selection. The **FMS_GETFILESEL** structure has the following form:

```
#include <wfext.h>

typedef struct tagFMS_GETFILESEL { /* fmsgfs */
    UINT  wTime;
    UINT  wDate;
    DWORD dwSize;
    BYTE  bAttr;
    char  szName[260];
} FMS_GETFILESEL;
```

Return Value

The return value is the zero-based index of the selected file that was retrieved.

Comments

Only extensions that support long filenames (for example, network-aware extensions) should use this message.

An extension can use the FM_GETSELCOUNT message to obtain the count of selected files.

The **szName** member of the **FMS_GETFILESEL** structure consists of an OEM character string. Before displaying this string, an extension should use the **OemToAnsi** function to convert the string to a Windows ANSI character string. If a string is to be passed to the file system (MS-DOS), an extension should not convert it.

See Also

FMExtensionProc, FM_GETFILESEL, FM_GETSELCOUNT, FM_GETSELCOUNTLFN, **OemToAnsi**

FM_GETFOCUS

A File Manager extension sends a FM_GETFOCUS message to retrieve the type of the File Manager window that has the input focus.

Parameters

wParam
Not used.

lParam
Not used.

Return Value

The return value indicates the type of File Manager window that has input focus. It can have one of the following values:

Value	Meaning
FMFOCUS_DIR	Directory portion of a directory window
FMFOCUS_TREE	Tree portion of a directory window
FMFOCUS_DRIVES	Drive bar of a directory window
FMFOCUS_SEARCH	Search Results window

FM_GETSELCOUNT

A File Manager extension sends a FM_GETSELCOUNT message to retrieve a count of the selected files in the directory or the Search Results window, depending on which is the active window.

Parameters

wParam
Not used.

lParam
Not used.

Return Value

The return value is the number of selected files.

See Also

FM_GETFILESEL, FM_GETFILESELLFN, FM_GETSELCOUNTLFN

FM_GETSELCOUNTLFN

A File Manager extension sends an FM_GETSELCOUNTLFN message to retrieve the number of selected files in the directory or the Search Results window, depending on which is the active window. The count includes files that have long filenames.

Parameters

wParam
Not used.

lParam
Not used.

Return Value The return value is the number of selected files.

Comments Only extensions that support long filenames (for example, network-aware extensions) should use this message.

See Also FM_GETFILESEL, FM_GETFILESELLFN, FM_GETSELCOUNT

FM_REFRESH_WINDOWS

A File Manager extension sends an FM_REFRESH_WINDOWS message to cause File Manager to repaint either its active window or all of its windows.

Parameters

wParam
Specifies whether File Manager repaints its active window or all of its windows. If this parameter is nonzero, File Manager repaints all of its windows. If this parameter is zero, File Manager repaints only its active window.

lParam
Not used.

Return Value This message does not return a meaningful value.

Comments File system changes caused by an extension are automatically detected by File Manager. An extension should use this message only in situations where drive connections are made or canceled.

See Also **FMExtensionProc**

FM_RELOAD_EXTENSIONS

A File Manager extension (or another application) sends an FM_RELOAD_EXTENSIONS message to cause File Manager to reload all extension dynamic-link libraries (DLLs) listed in the [AddOns] section of the WINFILE.INI file.

Parameters

wParam
Not used.

lParam
Not used.

Return Value

This message does not return a meaningful value.

Comments

Other applications can use the **PostMessage** function to send this message to File Manager. To obtain the appropriate File Manager window handle, an application can specify WFS_Frame as the *lpszClassName* parameter in a call to the **FindWindow** function.

See Also

FindWindow, **FMExtensionProc**, **PostMessage**

Control Panel Messages

Chapter 8

Control Panel communicates with a Control Panel dynamic-link library (DLL) through messages it sends to the **CPlApplet** entry-point function. A message consists of three parts: a message number and two 32-bit parameters. Message numbers are identified by predefined message names. The two 32-bit parameters contain message-dependent values.

This chapter contains an alphabetic list of all messages that can be received by the **CPlApplet** entry-point function. To use these messages, you must include the CPL.H header file.

CPL_DBLCLK

3.1

The CPL_DBLCLK message is sent to a Control Panel dynamic-link library (DLL) when the user double-clicks the icon of an application supported by the DLL.

Parameters

lParam1
Specifies the application number. This number must be in the range zero through one less than the value returned in response to the CPL_GETCOUNT message (CPL_GETCOUNT – 1).

lParam2
Specifies the value loaded into the **lData** member for the application.

Return Value

The Control Panel DLL returns zero if it processes this message successfully.

Comments

In response to this message, a Control Panel DLL should display the dialog box for the application.

See Also

CPL_GETCOUNT

CPL_EXIT

3.1

The CPL_EXIT message is sent once to a Control Panel dynamic-link library (DLL) before Control Panel calls the **FreeLibrary** function to free the DLL.

Parameters

lParam1
Not used.

lParam2
Not used.

Return Value

The Control Panel DLL returns zero if it processes this message successfully.

Comments

In response to this message, a Control Panel DLL should free any memory that it has allocated and perform global-level cleanup.

CPL_GETCOUNT

3.1

The CPL_GETCOUNT message retrieves the number of applications a Control Panel dynamic-link library (DLL) services.

Parameters

lParam1
Not used.

lParam2
Not used.

Return Value

The Control Panel DLL returns the number of applications it services.

Comments

This message is sent immediately after the CPL_INIT message.

See Also

CPL_INIT

CPL_INIT

3.1

The CPL_INIT message prompts a Control Panel dynamic-link library (DLL) to perform global initialization, especially memory allocation.

Parameters

lParam1
Not used.

lParam2
Not used.

Return Value

The Control Panel DLL returns nonzero if initialization is successful. Otherwise, it returns zero. If the DLL returns zero, Control Panel calls the **FreeLibrary** function and ends communication with the DLL.

Comments

Because this is the only way a Control Panel DLL can signal an error condition, the DLL should allocate memory in response to this message.

This message is sent immediately after the DLL is loaded.

CPL_INQUIRE

The CPL_INQUIRE message is sent to a Control Panel dynamic-link library (DLL) to request information about an application that the DLL supports.

This message is provided for backward compatibility with the Microsoft Windows operating system, version 3.1. An application should use the CPL_NEWINQUIRE message instead of the CPL_INQUIRE message.

Parameters

lParam1
Specifies the application number. This number must be in the range zero through one less than the value returned in response to the CPL_GETCOUNT message (CPL_GETCOUNT – 1).

lParam2
Specifies a far pointer to a **CPLINFO** structure. The DLL should fill this structure with resource identifiers for the icon, short name, description, and any user-defined value associated with the application. The **CPLINFO** structure has the following form:

```
#include <cpl.h>

typedef struct tagCPLINFO { /* cpli */
    int   idIcon;
    int   idName;
    int   idInfo;
    LONG  lData;
} CPLINFO;
```

Return Value

The Control Panel DLL returns zero if it processes this message successfully.

Comments

This message is sent once for each application serviced by the DLL. It is sent immediately after the CPL_GETCOUNT message. A DLL can perform application-level initialization when it receives this message. Memory should be allocated in response to the CPL_INIT message.

See Also

CPL_GETCOUNT, CPL_INIT, CPL_NEWINQUIRE

CPL_NEWINQUIRE

The CPL_NEWINQUIRE message is sent to a Control Panel dynamic-link library (DLL) to request information about an application that the DLL supports.

Parameters

lParam1
Specifies the application number. This number must be in the range zero through one less than the value returned in response to the CPL_GETCOUNT message (CPL_GETCOUNT – 1).

lParam2
Specifies a far pointer to a **NEWCPLINFO** structure. The DLL should fill this structure with information about the application. The **NEWCPLINFO** structure has the following form:

```
#include <cpl.h>

typedef struct tagNEWCPLINFO {  /* ncpli */
    DWORD   dwSize;
    DWORD   dwFlags;
    DWORD   dwHelpContext;
    LONG    lData;
    HICON   hIcon;
    char    szName[32];
    char    szInfo[64];
    char    szHelpFile[128];
} NEWCPLINFO;
```

Return Value

The Control Panel DLL returns zero if it processes this message successfully.

Comments

This message is sent once for each application serviced by the DLL. It is sent immediately after the CPL_GETCOUNT message. A DLL can use the switch block for this message to do application-level initialization when it receives this message. Memory should be allocated in response to the CPL_INIT message.

See Also

CPL_GETCOUNT, CPL_INIT, CPL_INQUIRE

CPL_SELECT

The CPL_SELECT message is sent to a Control Panel dynamic-link library (DLL) when the user selects the icon of an application supported by the DLL from Control Panel.

Parameters

lParam1
Specifies the application number.

lParam2
Specifies the application-defined value loaded in the **lData** member for the application.

Return Value

The Control Panel DLL returns zero if it processes this message successfully.

CPL_STOP

The CPL_STOP message is sent once for each application when Control Panel ends.

Parameters

lParam1
Specifies the application number. This number must be in the range zero through one less than the value returned in response to the CPL_GETCOUNT message (CPL_GETCOUNT – 1).

lParam2
Specifies the application-defined value loaded in the **lData** member for the application.

Return Value

The Control Panel DLL returns zero if it processes this message successfully.

Comments

In response to this message, a Control Panel DLL should perform application-specific cleanup.

See Also

CPL_GETCOUNT

WM_CPL_LAUNCH

An application sends the WM_CPL_LAUNCH message to Control Panel to request that a Control Panel application be started.

Parameters

wParam
Specifies the handle of the window sending the message. The WM_CPL_LAUNCHED message is sent to this window.

lParam
Specifies a far pointer to a string containing the name of the application to open.

Return Value

The return value is nonzero if the application was launched. Otherwise, it is zero.

Comments

The string referenced by the *lParam* parameter must be contained in a global memory object allocated with the GMEM_NOT_BANKED flag.

See Also

WM_CPL_LAUNCHED

WM_CPL_LAUNCHED

The WM_CPL_LAUNCHED message is sent when a Control Panel application, started by the WM_CPL_LAUNCH message, has ended. The WM_CPL_LAUNCHED message is sent to the window identified by the *wParam* parameter of the WM_CPL_LAUNCH message that started the application.

Parameters

wParam
Specifies whether the application was started. If the application was started, this parameter is nonzero. Otherwise, it is zero.

lParam
Not used.

Return Value

The value returned by the application is ignored for this message.

See Also

WM_CPL_LAUNCH

Common Dialog Box Messages

Chapter 9

A common dialog box sends a message to notify applications that the user has made or changed a selection in the dialog box. Applications can use these messages to carry out custom actions, such as rejecting certain user selections or setting custom colors.

Before an application can use a common dialog box message, it must register that message by using the **RegisterWindowMessage** function and the message constants given in this chapter and defined in the COMMDLG.H header file.

This chapter describes the common dialog box messages. The messages appear in alphabetic order.

COLOROKSTRING

3.1

The COLOROKSTRING message is sent by the Color dialog box to the application's hook function immediately before the dialog box is closed. This message allows more control over custom colors by giving the application the opportunity to leave the Color dialog box open when the user presses the OK button.

Parameters

wParam
Not used.

lParam
Points to a **CHOOSECOLOR** structure that specifies the currently selected color.

Return Value

If the application returns a nonzero value when it processes this message, the dialog box is not dismissed.

Comments

To use this message, the application must create a new message identifier by calling the **RegisterWindowMessage** function and passing the COLOROKSTRING constant as the single parameter.

See Also

RegisterWindowMessage

FILEOKSTRING

3.1

The FILEOKSTRING message is sent by the Open dialog box or Save As dialog box to the application's hook function when the user has selected a filename and chosen the OK button. The message lets the application accept or reject the user-selected filename.

Parameters

wParam
Not used.

lParam
Points to an **OPENFILENAME** structure containing information about the user's selection. (This information includes the filename for the selection.)

Return Value

The hook function should return 1 if it rejects the user-selected filename. In this case, the dialog box remains open and the user must select another filename. The hook function should return 0 if it accepts the user-selected filename or does not process the message.

Comments

To use this message, the application must create a message identifier by using the **RegisterWindowMessage** function and passing the FILEOKSTRING constant as the function's single parameter.

See Also

RegisterWindowMessage

FINDMSGSTRING

3.1

The FINDMSGSTRING message is sent to the application by the Find dialog box or Replace dialog box whenever the user has typed selections and chosen the OK button. This message contains data specified by the user in the dialog box controls, such as the direction in which the application should search for a string, whether the application should match the case of the specified string, or whether the application should match the string as an entire word.

Parameters

wParam
Not used.

lParam
Points to a **FINDREPLACE** structure containing information about the user's selections.

Return Value The application should return zero.

Comments To use the FINDMSGSTRING message, the application must create a message identifier by using the **RegisterWindowMessage** and passing the FINDMSGSTRING constant as the function's only parameter.

See Also **RegisterWindowMessage**

HELPMSGSTRING

3.1

The HELPMSGSTRING message is sent by a common dialog box to its owner's window procedure whenever the user chooses the Help button. This message lets an application provide custom Help for the common dialog boxes.

Parameters

wParam
Not used.

lParam
Points to the structure that describes the common dialog box.

Return Value The application returns zero.

Comments To use the HELPMSGSTRING message, the application must create a message identifier by using the **RegisterWindowMessage** function and passing the HELPMSGSTRING constant as the function's single parameter.

In addition to creating a new message identifier, the application must set the **hwndOwner** member in the appropriate data structure for the common dialog box. This member must contain the handle of the window to receive the HELPMSGSTRING message.

The application can also process the request for Help in a hook function. The hook function would identify this request by checking whether the *wParam* parameter of the WM_COMMAND message was equal to **psh 15**.

See Also **RegisterWindowMessage**

LBSELCHSTRING

3.1

The LBSELCHSTRING message is sent to an application's hook function by the Open or Save As dialog box whenever the user makes or changes a selection in the File Name list box. This message lets an application identify a new selection and carry out any application-specific actions, such as updating a custom control in the dialog box.

Parameters

wParam
Identifies the list box in which the selection occurred.

lParam
Identifies the list box item and type of selection. The low-order word of the *lParam* parameter identifies the list box item. The high-order word of the *lParam* parameter is one of the following values:

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

Return Value

The application returns zero.

Comments

To use the LBSELCHSTRING message, the application must create a message identifier by using the **RegisterWindowMessage** function and passing the LBSELCHSTRING constant as the function's single parameter.

See Also

RegisterWindowMessage

SETRGBSTRING

3.1

The SETRGBSTRING message is sent by an application's hook function to a Color dialog box to set a custom color.

Parameters

wParam
Not used.

lParam
Specifies the color to set. This parameter must be a red, green, blue (RGB) value.

Return Value

This message has no return value.

Comments

To use the SETRGBSTRING message, the application must create a message identifier by using the **RegisterWindowMessage** function and passing the SETRGBSTRING constant as the function's single parameter.

See Also

RegisterWindowMessage

SHAREVISTRING

3.1

The SHAREVISTRING message is sent to the application's hook function by the Open or Save As dialog box if a sharing violation occurs when the dialog box tries to open a file on the network.

Parameters

wParam
Not used.

lParam
Points to a string identifying the path and filename that caused the sharing violation. This string is the **szPathName** member of the **OFSTRUCT** structure that is pointed to by the second parameter of the **OpenFile** function.

Return Value

The return value is described in the following Comments section.

Comments

To use the SHAREVISTRING message, the application must create a message identifier by using the **RegisterWindowMessage** function and passing the SHAREVISTRING constant as the function's single parameter.

This message is sent by the **OpenFile** function. The message is not sent when the OFN_SHAREAWARE flag is set in the **Flags** member of the **OPENFILENAME** structure.

When the hook function receives SHAREVISTRING, it should return OFN_SHAREWARN, OFN_SHARENOWARN, or OFN_SHAREFALLTHROUGH. For more information about these flags, see the description of the **OPENFILENAME** structure in Chapter 3, "Structures."

See Also **OpenFile**, **RegisterWindowMessage**

Installable Driver Messages

Chapter 10

Installable driver messages notify installable drivers of specific events, such as loading or unloading the driver, or direct the driver to carry out some action, such as displaying a configuration dialog box. The Microsoft Windows operating system, version 3.1, sends installable driver messages to the **DriverProc** function of an installable driver whenever an application calls functions, such as **OpenDriver**, **SendDriverMessage**, and **CloseDriver**.

This chapter lists the installable driver messages in alphabetic order.

DRV_CLOSE

3.1

The DRV_CLOSE message is the first message sent by Windows to an installable driver after an application calls the **CloseDriver** function.

Parameters

dwDriverIdentifier
Specifies the unique 32-bit identifier returned by the **OpenDriver** function.

hDriver
Identifies the instance of the installable driver that should be closed.

lParam1
Specifies driver-specific data.

lParam2
Specifies driver-specific data.

Return Value

An installable driver returns nonzero if its **DriverProc** function successfully closes the driver. Otherwise, it returns zero.

Comments

The *lParam1* and *lParam2* parameters specify the same values as the *lParam1* and *lParam2* parameters for the **CloseDriver** function.

Each time a driver processes this message, it must decrement a private use-count variable. When the value of this variable is zero, Windows closes the driver.

See Also

DRV_OPEN

DRV_CONFIGURE

3.1

The DRV_CONFIGURE message is sent to inform an installable driver that it should display its private configuration dialog box.

Parameters

dwDriverIdentifier
Specifies a unique 32-bit value that identifies the installable driver.

hDriver
Identifies an instance of the installable driver.

lParam1
Specifies the handle of the parent window for the configuration dialog box. This handle is in the parameter's low-order word.

lParam2
Points to an optional **DRVCONFIGINFO** structure. An installable driver should verify that this pointer is valid before using it.

This structure has the following form:

```
typedef struct tagDRVCONFIGINFO {
    DWORD   dwDCISize;
    LPCSTR  lpszDCISectionName;
    LPCSTR  lpszDCIAliasName;
} DRVCONFIGINFO;
```

Return Value

An installable driver returns nonzero if it processes this message. Otherwise, it returns zero.

Comments

An installable driver that supports the DRV_CONFIGURE message must provide its own dialog box template and dialog box procedure. It must also record the user's configuration requests in an appropriate file. (This may be the SYSTEM.INI file or some other file used by the driver for this purpose.)

See Also

DRV_QUERYCONFIGURE

DRV_DISABLE

3.1

The DRV_DISABLE message is the second message sent by Windows to an installable driver after an application calls the **CloseDriver** function.

Parameters

dwDriverIdentifier
Not used.

hDriver
Identifies an instance of the installable driver.

lParam1
Not used.

lParam2
Not used.

Return Value

An installable driver returns zero if it processes this message.

See Also

DRV_CLOSE

DRV_ENABLE

3.1

The DRV_ENABLE message is sent to an installable driver when it is loaded or reloaded or whenever Windows is reinstalled after switching to an MS-DOS application.

Parameters

dwDriverIdentifier
Not used.

hDriver
Identifies an instance of the installable driver.

lParam1
Not used.

lParam2
Not used.

Return Value An installable driver returns zero if it processes this message.

Comments When the **DriverProc** function receives this message, it should initialize all of the driver-specific structures with default values.

See Also DRV_OPEN

DRV_EXITAPPLICATION

3.1

The DRV_EXITAPPLICATION message is sent to all installable drivers when an application exits.

Parameters

dwDriverIdentifier
Specifies a unique 32-bit value that identifies the installable driver.

lParam1
Specifies the type of application exit. This parameter can be one of the following values:

Value	Meaning
DRVEA_NORMALEXIT	Set if the application terminated normally.
DRVEA_ABNORMALEXIT	Set if the application terminated abnormally (because of an application or system error).

lParam2
Not used.

Return Value The value returned by the application is ignored for this message.

See Also DRV_EXITSESSION

DRV_EXITSESSION

3.1

The DRV_EXITSESSION message is sent to all installable drivers when Windows prepares to exit.

Parameters

dwDriverIdentifier
Specifies a unique 32-bit value that identifies the installable driver.

lParam1
Reserved.

lParam2
Reserved.

Return Value

The value returned by the application is ignored for this message.

Comments

The user interface and all other drivers are still enabled when this message is sent.

See Also

DRV_EXITAPPLICATION

DRV_FREE

3.1

The DRV_FREE message is the third message sent by Windows to an installable driver after an application calls the **CloseDriver** function.

Parameters

dwDriverIdentifier
Not used.

hDriver
Identifies an instance of the installable driver.

lParam1
Not used.

lParam2
Not used.

Return Value

An installable driver returns zero if it processes this message.

Comments

When an installable driver's **DriverProc** function receives this message, it should free the memory that was allocated for all driver-specific structures.

DRV_INSTALL

3.1

The DRV_INSTALL message is sent to an installable driver during the driver initialization process.

Parameters

dwDriverIdentifier
Specifies a unique 32-bit value that identifies the installable driver.

hDriver
Identifies an instance of the installable driver.

lParam1
Not used.

lParam2
Points to an optional **DRVCONFIGINFO** structure. An installable driver should verify that this pointer is valid before using it.

This structure has the following form:

```
typedef struct tagDRVCONFIGINFO {
    DWORD   dwDCISize;
    LPCSTR  lpszDCISectionName;
    LPCSTR  lpszDCIAliasName;
} DRVCONFIGINFO;
```

Return Value

An installable driver returns nonzero if it processes this message. Otherwise, it returns zero.

Comments

When the driver receives this message, it creates an entry for the driver in the SYSTEM.INI file and performs other necessary configuration operations.

DRV_LOAD

3.1

The DRV_LOAD message is sent to an installable driver to notify the driver that it has been loaded.

Parameters

dwDriverIdentifier
Not used.

hDriver
Identifies an instance of the installable driver.

lParam1
Not used.

lParam2
Not used.

Return Value

An installable driver returns nonzero if its **DriverProc** function successfully loads the driver. Otherwise, it returns zero.

DRV_OPEN

3.1

The DRV_OPEN message is sent to an installable driver each time it is opened.

Parameters

dwDriverIdentifier
Specifies a unique 32-bit value that identifies the installable driver.

hDriver
Identifies an instance of the installable driver.

lParam1
Points to a null-terminated string containing any ASCII characters that followed the driver name in the SYSTEM.INI file.

lParam2
Contains the data specified by the *lParam* parameter, the third argument in the **OpenDriver** function.

Return Value

An installable driver returns nonzero if it processes this message. Otherwise, it returns zero.

Comments

If no characters follow the driver name in SYSTEM.INI, the *lParam1* parameter is a NULL pointer.

See Also

DRV_CLOSE

DRV_POWER

3.1

The DRV_POWER message is sent to an installable driver each time the power supply to the associated device is about to be turned on or off.

Parameters

dwDriverIdentifier
Specifies a unique 32-bit value that identifies the installable driver.

hDriver
Identifies an instance of the installable driver.

lParam1
Not used.

lParam2
Not used.

Return Value

An installable driver returns nonzero if it processes this message. Otherwise, it returns zero.

DRV_QUERYCONFIGURE

3.1

The DRV_QUERYCONFIGURE message is sent to an installable driver to determine whether it can be configured by the user.

Parameters

dwDriverIdentifier
Specifies a unique 32-bit value that identifies the installable driver.

hDriver
Identifies an instance of the installable driver.

lParam1
Not used.

lParam2
Not used.

Return Value

An installable driver returns nonzero if it supports custom configuration and is capable of displaying a configuration dialog box. Otherwise, it returns zero.

See Also

DRV_CONFIGURE

DRV_REMOVE

3.1

The DRV_REMOVE message is sent by an application to an installable driver to notify the driver that it is about to be removed from the system.

Parameters

dwDriverIdentifier
Specifies a unique 32-bit value that identifies the installable driver.

lParam1
Not used.

lParam2
Not used.

Return Value

An installable driver returns nonzero if it processes this message. Otherwise, it returns zero.

Comments

When an installable driver receives this message, it should remove necessary entries from the SYSTEM.INI file.

DRV_USER

3.1

The DRV_USER message is a user-defined or driver-dependent message.

Parameters

dwDriverIdentifier
This parameter is not predefined; the value is driver dependent.

hDriver
This parameter is not predefined; the value is driver dependent.

lParam1
This parameter is not predefined; the value is driver dependent.

lParam2
This parameter is not predefined; the value is driver dependent.

Return Value

The return value is driver dependent.

Binary and Ternary Raster-Operation Codes

Appendix A

This appendix lists and describes the binary and ternary raster operations used by graphics device interface (GDI). A binary raster operation involves two operands: a pen and a destination bitmap. A ternary raster operation involves three operands: a source bitmap, a brush, and a destination bitmap. Both binary and ternary raster operations use Boolean operators.

A.1 Binary Raster Operations

This section lists the binary raster-operation codes used by the **GetROP2** and **SetROP2** functions. Raster-operation codes define how GDI combines the bits from the selected pen with the bits in the destination bitmap.

Each raster-operation code represents a Boolean operation in which the values of the pixels in the selected pen and the destination bitmap are combined. Following are the two operands used in these operations:

Operand	Meaning
P	Selected pen
D	Destination bitmap

The Boolean operators used in these operations follow:

Operator	Meaning
a	Bitwise AND
n	Bitwise NOT (inverse)
o	Bitwise OR
x	Bitwise exclusive OR (XOR)

All Boolean operations are presented in reverse Polish notation. For example, the following operation replaces the values of the pixels in the destination bitmap with a combination of the pixel values of the pen and the selected brush:

```
DPo
```

Each raster-operation code is a 32-bit integer whose high-order word is a Boolean operation index and whose low-order word is the operation code. The 16-bit operation index is a zero-extended 8-bit value that represents all possible outcomes

resulting from the Boolean operation on two parameters (in this case, the pen and destination values). For example, the operation indexes for the DPo and DPan operations are shown in the following list:

P	D	DPo	DPan
0	0	0	1
0	1	1	1
1	0	1	1
1	1	1	0

The following list outlines the drawing modes and the Boolean operations that they represent:

Raster operation	Boolean operation
R2_BLACK	0
R2_COPYPEN	P
R2_MASKNOTPEN	DPna
R2_MASKPEN	DPa
R2_MASKPENNOT	PDna
R2_MERGENOTPEN	DPno
R2_MERGEPEN	DPo
R2_MERGEPENNOT	PDno
R2_NOP	D
R2_NOT	Dn
R2_NOTCOPYPEN	Pn
R2_NOTMASKPEN	DPan
R2_NOTMERGEPEN	DPon
R2_NOTXORPEN	DPxn
R2_WHITE	1
R2_XORPEN	DPx

For a monochrome device, GDI maps the value zero to black and the value 1 to white. If an application attempts to draw with a black pen on a white destination by using the available binary raster operations, the following results occur:

Raster operation	Result
R2_BLACK	Visible black line
R2_COPYPEN	Visible black line
R2_MASKNOTPEN	No visible line
R2_MASKPEN	Visible black line
R2_MASKPENNOT	Visible black line

Raster operation	Result
R2_MERGENOTPEN	No visible line
R2_MERGEPEN	Visible black line
R2_MERGEPENNOT	Visible black line
R2_NOP	No visible line
R2_NOT	Visible black line
R2_NOTCOPYPEN	No visible line
R2_NOTMASKPEN	No visible line
R2_NOTMERGEPEN	Visible black line
R2_NOTXORPEN	Visible black line
R2_WHITE	No visible line
R2_XORPEN	No visible line

For a color device, GDI uses RGB values to represent the colors of the pen and the destination. An RGB color value is a long integer that contains a red, a green, and a blue color field, each specifying the intensity of the given color. Intensities range from 0 through 255. The values are packed in the three low-order bytes of the long integer. The color of a pen is always a solid color, but the color of the destination may be a mixture of any two or three colors. If an application attempts to draw with a white pen on a blue destination by using the available binary raster operations, the following results occur:

Raster operation	Result
R2_BLACK	Visible black line
R2_COPYPEN	Visible white line
R2_MASKNOTPEN	Visible black line
R2_MASKPEN	Invisible blue line
R2_MASKPENNOT	Visible red/green line
R2_MERGENOTPEN	Invisible blue line
R2_MERGEPEN	Visible white line
R2_MERGEPENNOT	Visible white line
R2_NOP	Invisible blue line
R2_NOT	Visible red/green line
R2_NOTCOPYPEN	Visible black line
R2_NOTMASKPEN	Visible red/green line
R2_NOTMERGEPEN	Visible black line
R2_NOTXORPEN	Invisible blue line
R2_WHITE	Visible white line
R2_XORPEN	Visible red/green line

A.2 Ternary Raster Operations

This section lists the ternary raster-operation codes used by the **BitBlt**, **PatBlt**, and **StretchBlt** functions. Ternary raster-operation codes define how GDI combines the bits in a source bitmap with the bits in the destination bitmap.

Each raster-operation code represents a Boolean operation in which the values of the pixels in the source, the selected brush, and the destination are combined. Following are the three operands used in these operations:

Operand	Meaning
D	Destination bitmap
P	Selected brush (also called pattern)
S	Source bitmap

Boolean operators used in these operations follow:

Operator	Meaning
a	Bitwise AND
n	Bitwise NOT (inverse)
o	Bitwise OR
x	Bitwise exclusive OR (XOR)

All Boolean operations are presented in reverse Polish notation. For example, the following operation replaces the values of the pixels in the destination bitmap with a combination of the pixel values of the source and brush:

```
PSo
```

The following operation combines the values of the pixels in the source and brush with the pixel values of the destination bitmap (there are alternative spellings of the same function, so although a particular spelling may not be in the list, an equivalent form would be):

```
DPSoo
```

Each raster-operation code is a 32-bit integer whose high-order word is a Boolean operation index and whose low-order word is the operation code. The 16-bit operation index is a zero-extended, 8-bit value that represents the result of the Boolean operation on predefined brush, source, and destination values. For example, the operation indexes for the PSo and DPSoo operations are shown in the following list:

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:			00FCh	00FEh

In this case, PSo has the operation index 00FC (read from the bottom up); DPSoo has the operation index 00FE. These values define the location of the corresponding raster-operation codes, as shown in Table A.1, "Raster-Operation Codes." The PSo operation is in line 252 (00FCh) of the table; DPSoo is in line 254 (00FEh).

The most commonly used raster operations have been given special names in the Windows include file, WINDOWS.H. You should use these names whenever possible in your applications.

When the source and destination bitmaps are monochrome, a bit value of zero represents a black pixel and a bit value of 1 represents a white pixel. When the source and the destination bitmaps are color, those colors are represented with RGB values. For more information about RGB values, see the **RGB** structure in Chapter 3, "Structures."

Table A.1 Raster-Operation Codes

Boolean function (hexadecimal)	Raster operation (hexadecimal)	Boolean function in reverse Polish	Common name
00	00000042	0	BLACKNESS
01	00010289	DPSoon	–
02	00020C89	DPSona	–
03	000300AA	PSon	–
04	00040C88	SDPona	–
05	000500A9	DPon	–
06	00060865	PDSxnon	–
07	000702C5	PDSaon	–
08	00080F08	SDPnaa	–
09	00090245	PDSxon	–
0A	000A0329	DPna	–
0B	000B0B2A	PSDnaon	–
0C	000C0324	SPna	–
0D	000D0B25	PDSnaon	–
0E	000E08A5	PDSonon	–
0F	000F0001	Pn	–
10	00100C85	PDSona	–
11	001100A6	DSon	NOTSRCERASE
12	00120868	SDPxnon	–
13	001302C8	SDPaon	–
14	00140869	DPSxnon	–
15	001502C9	DPSaon	–
16	00165CCA	PSDPSanaxx	–
17	00171D54	SSPxDSxaxn	–
18	00180D59	SPxPDxa	–
19	00191CC8	SDPSanaxn	–
1A	001A06C5	PDSPaox	–
1B	001B0768	SDPSxaxn	–
1C	001C06CA	PSDPaox	–
1D	001D0766	DSPDxaxn	–
1E	001E01A5	PDSox	–
1F	001F0385	PDSoan	–
20	00200F09	DPSnaa	–
21	00210248	SDPxon	–
22	00220326	DSna	–

Table A.1 Raster-Operation Codes *(continued)*

Boolean function (hexadecimal)	Raster operation (hexadecimal)	Boolean function in reverse Polish	Common name
23	00230B24	SPDnaon	–
24	00240D55	SPxDSxa	–
25	00251CC5	PDSPanaxn	–
26	002606C8	SDPSaox	–
27	00271868	SDPSxnox	–
28	00280369	DPSxa	–
29	002916CA	PSDPSaoxxn	–
2A	002A0CC9	DPSana	–
2B	002B1D58	SSPxPDxaxn	–
2C	002C0784	SPDSoax	–
2D	002D060A	PSDnox	–
2E	002E064A	PSDPxox	–
2F	002F0E2A	PSDnoan	–
30	0030032A	PSna	–
31	00310B28	SDPnaon	–
32	00320688	SDPSoox	–
33	00330008	Sn	NOTSRCCOPY
34	003406C4	SPDSaox	–
35	00351864	SPDSxnox	–
36	003601A8	SDPox	–
37	00370388	SDPoan	–
38	0038078A	PSDPoax	–
39	00390604	SPDnox	–
3A	003A0644	SPDSxox	–
3B	003B0E24	SPDnoan	–
3C	003C004A	PSx	–
3D	003D18A4	SPDSonox	–
3E	003E1B24	SPDSnaox	–
3F	003F00EA	PSan	–
40	00400F0A	PSDnaa	–
41	00410249	DPSxon	–
42	00420D5D	SDxPDxa	–
43	00431CC4	SPDSanaxn	–
44	00440328	SDna	SRCERASE
45	00450B29	DPSnaon	–

Table A.1 Raster-Operation Codes *(continued)*

Boolean function (hexadecimal)	Raster operation (hexadecimal)	Boolean function in reverse Polish	Common name
46	004606C6	DSPDaox	–
47	0047076A	PSDPxaxn	–
48	00480368	SDPxa	–
49	004916C5	PDSPDaoxxn	–
4A	004A0789	DPSDoax	–
4B	004B0605	PDSnox	–
4C	004C0CC8	SDPana	–
4D	004D1954	SSPxDSxoxn	–
4E	004E0645	PDSPxox	–
4F	004F0E25	PDSnoan	–
50	00500325	PDna	–
51	00510B26	DSPnaon	–
52	005206C9	DPSDaox	–
53	00530764	SPDSxaxn	–
54	005408A9	DPSonon	–
55	00550009	Dn	DSTINVERT
56	005601A9	DPSox	–
57	00570389	DPSoan	–
58	00580785	PDSPoax	–
59	00590609	DPSnox	–
5A	005A0049	DPx	PATINVERT
5B	005B18A9	DPSDonox	–
5C	005C0649	DPSDxox	–
5D	005D0E29	DPSnoan	–
5E	005E1B29	DPSDnaox	–
5F	005F00E9	DPan	–
60	00600365	PDSxa	–
61	006116C6	DSPDSaoxxn	–
62	00620786	DSPDoax	–
63	00630608	SDPnox	–
64	00640788	SDPSoax	–
65	00650606	DSPnox	–
66	00660046	DSx	SRCINVERT
67	006718A8	SDPSonox	–
68	006858A6	DSPDSonoxxn	–

Table A.1 Raster-Operation Codes *(continued)*

Boolean function (hexadecimal)	Raster operation (hexadecimal)	Boolean function in reverse Polish	Common name
69	00690145	PDSxxn	–
6A	006A01E9	DPSax	–
6B	006B178A	PSDPSoaxxn	–
6C	006C01E8	SDPax	–
6D	006D1785	PDSPDoaxxn	–
6E	006E1E28	SDPSnoax	–
6F	006F0C65	PDSxnan	–
70	00700CC5	PDSana	–
71	00711D5C	SSDxPDxaxn	–
72	00720648	SDPSxox	–
73	00730E28	SDPnoan	–
74	00740646	DSPDxox	–
75	00750E26	DSPnoan	–
76	00761B28	SDPSnaox	–
77	007700E6	DSan	–
78	007801E5	PDSax	–
79	00791786	DSPDSoaxxn	–
7A	007A1E29	DPSDnoax	–
7B	007B0C68	SDPxnan	–
7C	007C1E24	SPDSnoax	–
7D	007D0C69	DPSxnan	–
7E	007E0955	SPxDSxo	–
7F	007F03C9	DPSaan	–
80	008003E9	DPSaa	–
81	00810975	SPxDSxon	–
82	00820C49	DPSxna	–
83	00831E04	SPDSnoaxn	–
84	00840C48	SDPxna	–
85	00851E05	PDSPnoaxn	–
86	008617A6	DSPDSoaxx	–
87	008701C5	PDSaxn	–
88	008800C6	DSa	SRCAND
89	00891B08	SDPSnaoxn	–
8A	008A0E06	DSPnoa	–
8B	008B0666	DSPDxoxn	–

Table A.1 Raster-Operation Codes *(continued)*

Boolean function (hexadecimal)	Raster operation (hexadecimal)	Boolean function in reverse Polish	Common name
8C	008C0E08	SDPnoa	–
8D	008D0668	SDPSxoxn	–
8E	008E1D7C	SSDxPDxax	–
8F	008F0CE5	PDSanan	–
90	00900C45	PDSxna	–
91	00911E08	SDPSnoaxn	–
92	009217A9	DPSDPoaxx	–
93	009301C4	SPDaxn	–
94	009417AA	PSDPSoaxx	–
95	009501C9	DPSaxn	–
96	00960169	DPSxx	–
97	0097588A	PSDPSonoxx	–
98	00981888	SDPSonoxn	–
99	00990066	DSxn	–
9A	009A0709	DPSnax	–
9B	009B07A8	SDPSoaxn	–
9C	009C0704	SPDnax	–
9D	009D07A6	DSPDoaxn	–
9E	009E16E6	DSPDSaoxx	–
9F	009F0345	PDSxan	–
A0	00A000C9	DPa	–
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	–

Table A.1 Raster-Operation Codes *(continued)*

Boolean function (hexadecimal)	Raster operation (hexadecimal)	Boolean function in reverse Polish	Common name
AF	00AF0229	DPno	–
B0	00B00E05	PDSnoa	–
B1	00B10665	PDSPxoxn	–
B2	00B21974	SSPxDSxox	–
B3	00B30CE8	SDPanan	–
B4	00B4070A	PSDnax	–
B5	00B507A9	DPSDoaxn	–
B6	00B616E9	DPSDPaoxx	–
B7	00B70348	SDPxan	–
B8	00B8074A	PSDPxax	–
B9	00B906E6	DSPDaoxn	–
BA	00BA0B09	DPSnao	–
BB	00BB0226	DSno	MERGEPAINT
BC	00BC1CE4	SPDSanax	–
BD	00BD0D7D	SDxPDxan	–
BE	00BE0269	DPSxo	–
BF	00BF08C9	DPSano	–
C0	00C000CA	PSa	MERGECOPY
C1	00C11B04	SPDSnaoxn	–
C2	00C21884	SPDSonoxn	–
C3	00C3006A	PSxn	–
C4	00C40E04	SPDnoa	–
C5	00C50664	SPDSxoxn	–
C6	00C60708	SDPnax	–
C7	00C707AA	PSDPoaxn	–
C8	00C803A8	SDPoa	–
C9	00C90184	SPDoxn	–
CA	00CA0749	DPSDxax	–
CB	00CB06E4	SPDSaoxn	–
CC	00CC0020	S	SRCCOPY
CD	00CD0888	SDPono	–
CE	00CE0B08	SDPnao	–
CF	00CF0224	SPno	–
D0	00D00E0A	PSDnoa	–
D1	00D1066A	PSDPxoxn	–

Table A.1 Raster-Operation Codes *(continued)*

Boolean function (hexadecimal)	Raster operation (hexadecimal)	Boolean function in reverse Polish	Common name
D2	00D20705	PDSnax	–
D3	00D307A4	SPDSoaxn	–
D4	00D41D78	SSPxPDxax	–
D5	00D50CE9	DPSanan	–
D6	00D616EA	PSDPSaoxx	–
D7	00D70349	DPSxan	–
D8	00D80745	PDSPxax	–
D9	00D906E8	SDPSaoxn	–
DA	00DA1CE9	DPSDanax	–
DB	00DB0D75	SPxDSxan	–
DC	00DC0B04	SPDnao	–
DD	00DD0228	SDno	–
DE	00DE0268	SDPxo	–
DF	00DF08C8	SDPano	–
E0	00E003A5	PDSoa	–
E1	00E10185	PDSoxn	–
E2	00E20746	DSPDxax	–
E3	00E306EA	PSDPaoxn	–
E4	00E40748	SDPSxax	–
E5	00E506E5	PDSPaoxn	–
E6	00E61CE8	SDPSanax	–
E7	00E70D79	SPxPDxan	–
E8	00E81D74	SSPxDSxax	–
E9	00E95CE6	DSPDSanaxxn	–
EA	00EA02E9	DPSao	–
EB	00EB0849	DPSxno	–
EC	00EC02E8	SDPao	–
ED	00ED0848	SDPxno	–
EE	00EE0086	DSo	SRCPAINT
EF	00EF0A08	SDPnoo	–
F0	00F00021	P	PATCOPY
F1	00F10885	PDSono	–
F2	00F20B05	PDSnao	–
F3	00F3022A	PSno	–
F4	00F40B0A	PSDnao	–

Table A.1 Raster-Operation Codes *(continued)*

Boolean function (hexadecimal)	Raster operation (hexadecimal)	Boolean function in reverse Polish	Common name
F5	00F50225	PDno	–
F6	00F60265	PDSxo	–
F7	00F708C5	PDSano	–
F8	00F802E5	PDSao	–
F9	00F90845	PDSxno	–
FA	00FA0089	DPo	–
FB	00FB0A09	DPSnoo	PATPAINT
FC	00FC008A	PSo	–
FD	00FD0A0A	PSDnoo	–
FE	00FE02A9	DPSoo	–
FF	00FF0062	1	WHITENESS

Virtual-Key Codes

Appendix B

The following table shows the symbolic constant names, hexadecimal values, and keyboard equivalents for the virtual-key codes used by the Microsoft Windows operating system version 3.1. The codes are listed in numeric order.

Symbolic constant name	Value (hexadecimal)	Mouse or keyboard equivalent
VK_LBUTTON	01	Left mouse button
VK_RBUTTON	02	Right mouse button
VK_CANCEL	03	Used for control-break processing
VK_MBUTTON	04	Middle mouse button (three-button mouse)
–	05–07	Undefined
VK_BACK	08	BACKSPACE key
VK_TAB	09	TAB key
	0A–0B	Undefined
VK_CLEAR	0C	CLEAR key
VK_RETURN	0D	ENTER key
–	0E–0F	Undefined
VK_SHIFT	10	SHIFT key
VK_CONTROL	11	CTRL key
VK_MENU	12	ALT key
VK_PAUSE	13	PAUSE key
VK_CAPITAL	14	CAPS LOCK key
–	15–19	Reserved for Kanji systems
–	1A	Undefined
VK_ESCAPE	1B	ESC key
–	1C–1F	Reserved for Kanji systems
VK_SPACE	20	SPACEBAR
VK_PRIOR	21	PAGE UP key
VK_NEXT	22	PAGE DOWN key
VK_END	23	END key
VK_HOME	24	HOME key
VK_LEFT	25	LEFT ARROW key
VK_UP	26	UP ARROW key
VK_RIGHT	27	RIGHT ARROW key
VK_DOWN	28	DOWN ARROW key
VK_SELECT	29	SELECT key
–	2A	OEM specific
VK_EXECUTE	2B	EXECUTE key

Symbolic constant name	Value (hexadecimal)	Mouse or keyboard equivalent
VK_SNAPSHOT	2C	PRINT SCREEN key for Windows 3.0 and later
VK_INSERT	2D	INS key
VK_DELETE	2E	DEL key
VK_HELP	2F	HELP key
VK_0	30	0 key
VK_1	31	1 key
VK_2	32	2 key
VK_3	33	3 key
VK_4	34	4 key
VK_5	35	5 key
VK_6	36	6 key
VK_7	37	7 key
VK_8	38	8 key
VK_9	39	9 key
–	3A–40	Undefined
VK_A	41	A key
VK_B	42	B key
VK_C	43	C key
VK_D	44	D key
VK_E	45	E key
VK_F	46	F key
VK_G	47	G key
VK_H	48	H key
VK_I	49	I key
VK_J	4A	J key
VK_K	4B	K key
VK_L	4C	L key
VK_M	4D	M key
VK_N	4E	N key
VK_O	4F	O key
VK_P	50	P key
VK_Q	51	Q key
VK_R	52	R key
VK_S	53	S key
VK_T	54	T key
VK_U	55	U key

Symbolic constant name	Value (hexadecimal)	Mouse or keyboard equivalent
VK_V	56	V key
VK_W	57	W key
VK_X	58	X key
VK_Y	59	Y key
VK_Z	5A	Z key
–	5B–5F	Undefined
VK_NUMPAD0	60	Numeric keypad 0 key
VK_NUMPAD1	61	Numeric keypad 1 key
VK_NUMPAD2	62	Numeric keypad 2 key
VK_NUMPAD3	63	Numeric keypad 3 key
VK_NUMPAD4	64	Numeric keypad 4 key
VK_NUMPAD5	65	Numeric keypad 5 key
VK_NUMPAD6	66	Numeric keypad 6 key
VK_NUMPAD7	67	Numeric keypad 7 key
VK_NUMPAD8	68	Numeric keypad 8 key
VK_NUMPAD9	69	Numeric keypad 9 key
VK_MULTIPLY	6A	Multiply key
VK_ADD	6B	Add key
VK_SEPARATOR	6C	Separator key
VK_SUBTRACT	6D	Subtract key
VK_DECIMAL	6E	Decimal key
VK_DIVIDE	6F	Divide key
VK_F1	70	F1 key
VK_F2	71	F2 key
VK_F3	72	F3 key
VK_F4	73	F4 key
VK_F5	74	F5 key
VK_F6	75	F6 key
VK_F7	76	F7 key
VK_F8	77	F8 key
VK_F9	78	F9 key
VK_F10	79	F10 key
VK_F11	7A	F11 key
VK_F12	7B	F12 key
VK_F13	7C	F13 key
VK_F14	7D	F14 key
VK_F15	7E	F15 key

Symbolic constant name	Value (hexadecimal)	Mouse or keyboard equivalent
VK_F16	7F	F16 key
VK_F17	80H	F17 key
VK_F18	81H	F18 key
VK_F19	82H	F19 key
VK_F20	83H	F20 key
VK_F21	84H	F21 key
VK_F22	85H	F22 key
VK_F23	86H	F23 key
VK_F24	87H	F24 key
–	88–8F	Unassigned
VK_NUMLOCK	90	NUM LOCK key
VK_SCROLL	91	SCROLL LOCK key
–	92–B9	Unassigned
–	BA–C0	OEM specific
–	C1–DA	Unassigned
–	DB–E4	OEM specific
–	E5	Unassigned
–	E6	OEM specific
–	E7–E8	Unassigned
–	E9–F5	OEM specific
–	F6–FE	Unassigned

Character Sets

Appendix C

The Microsoft Windows operating system, version 3.1 supports multiple character sets, allowing for customization. Among the character sets that Windows 3.1 provides are the Windows, Symbol, and OEM character sets, shown in the following sections.

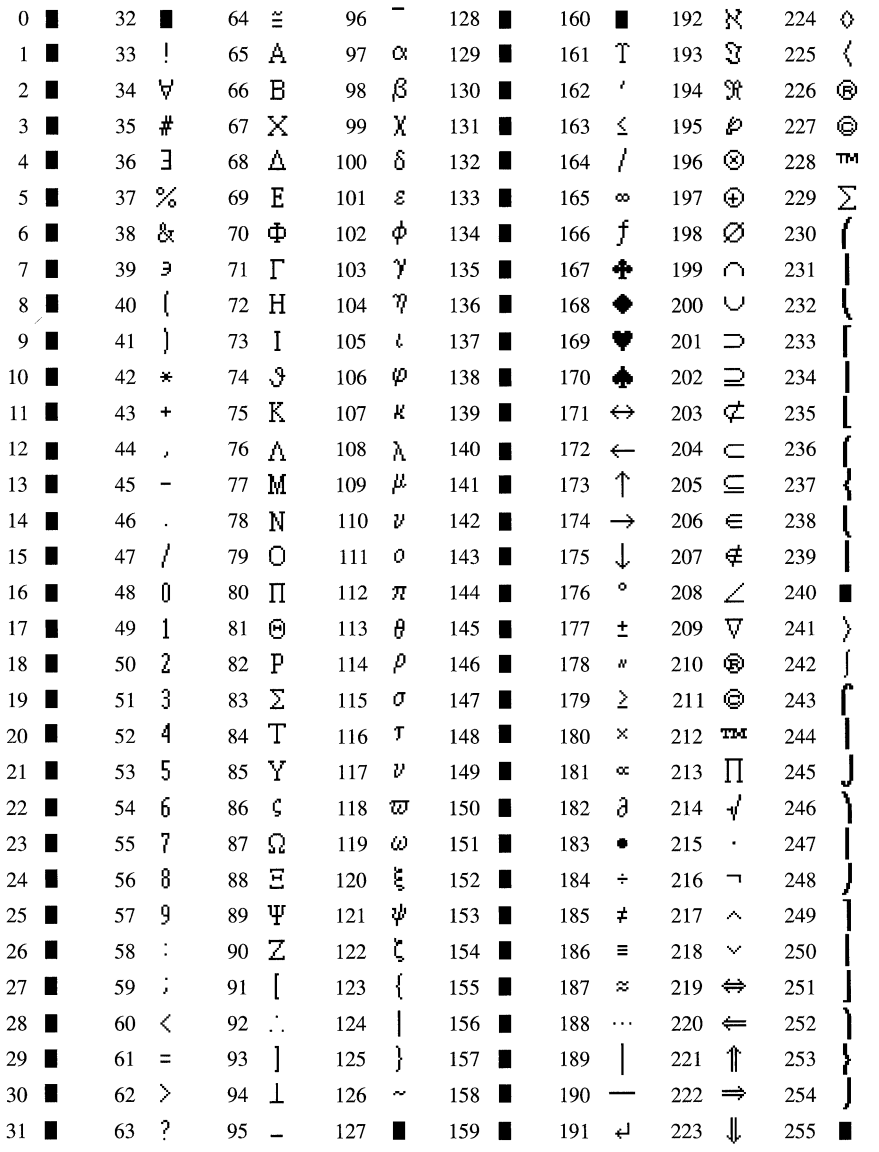
C.1 ANSI Character Set

0	■	32		64	@	96	`	128	■	160		192	À	224	à
1	■	33	!	65	A	97	a	129	■	161	¡	193	Á	225	á
2	■	34	"	66	B	98	b	TT 130	‚	162	¢	194	Â	226	â
3	■	35	#	67	C	99	c	TT 131	*f*	163	£	195	Ã	227	ã
4	■	36	$	68	D	100	d	TT 132	„	164	¤	196	Ä	228	ä
5	■	37	%	69	E	101	e	TT 133	…	165	¥	197	Å	229	å
6	■	38	&	70	F	102	f	TT 134	†	166	¦	198	Æ	230	æ
7	■	39	'	71	G	103	g	TT 135	‡	167	§	199	Ç	231	ç
8	■	40	(	72	H	104	h	TT 136	ˆ	168	¨	200	È	232	è
9	■	41	)	73	I	105	i	TT 137	‰	169	©	201	É	233	é
10	■	42	*	74	J	106	j	TT 138	Š	170	ª	202	Ê	234	ê
11	■	43	+	75	K	107	k	TT 139	‹	171	«	203	Ë	235	ë
12	■	44	,	76	L	108	l	TT 140	Œ	172	¬	204	Ì	236	ì
13	■	45	-	77	M	109	m	141	■	173	-	205	Í	237	í
14	■	46	.	78	N	110	n	142	■	174	®	206	Î	238	î
15	■	47	/	79	O	111	o	143	■	175	¯	207	Ï	239	ï
16	■	48	0	80	P	112	p	144	■	176	°	208	Ð	240	ð
17	■	49	1	81	Q	113	q	145	‘	177	±	209	Ñ	241	ñ
18	■	50	2	82	R	114	r	146	’	178	²	210	Ò	242	ò
19	■	51	3	83	S	115	s	TT 147	“	179	³	211	Ó	243	ó
20	■	52	4	84	T	116	t	TT 148	”	180	´	212	Ô	244	ô
21	■	53	5	85	U	117	u	TT 149	•	181	µ	213	Õ	245	õ
22	■	54	6	86	V	118	v	TT 150	–	182	¶	214	Ö	246	ö
23	■	55	7	87	W	119	w	TT 151	—	183	·	215	×	247	÷
24	■	56	8	88	X	120	x	TT 152	˜	184	¸	216	Ø	248	ø
25	■	57	9	89	Y	121	y	TT 153	™	185	¹	217	Ù	249	ù
26	■	58	:	90	Z	122	z	TT 154	š	186	º	218	Ú	250	ú
27	■	59	;	91	[	123	{	TT 155	›	187	»	219	Û	251	û
28	■	60	<	92	\	124	|	TT 156	œ	188	¼	220	Ü	252	ü
29	■	61	=	93	]	125	}	157	■	189	½	221	Ý	253	ý
30	■	62	>	94	^	126	~	158	■	190	¾	222	Þ	254	þ
31	■	63	?	95	_	127	■	TT 159	Ÿ	191	¿	223	ß	255	ÿ

■ Indicates that this character is not supported by Windows.

TT Indicates that this character is available only in TrueType fonts.

C.2 Symbol Character Set



■ Indicates that this character is not supported by Windows.

C.3 OEM Character Set

0		32		64	@	96	`	128	Ç	160	á	192	└	224	α
1	☺	33	!	65	A	97	a	129	ü	161	í	193	┴	225	ß
2	☻	34	"	66	B	98	b	130	é	162	ó	194	┬	226	Γ
3	♥	35	#	67	C	99	c	131	â	163	ú	195	├	227	π
4	♦	36	$	68	D	100	d	132	ä	164	ñ	196	─	228	Σ
5	♣	37	%	69	E	101	e	133	à	165	Ñ	197	┼	229	σ
6	♠	38	&	70	F	102	f	134	å	166	ª	198	╞	230	µ
7	•	39	'	71	G	103	g	135	ç	167	º	199	╟	231	τ
8	◘	40	(	72	H	104	h	136	ê	168	¿	200	╚	232	Φ
9	○	41	)	73	I	105	i	137	ë	169	⌐	201	╔	233	Θ
10	◙	42	*	74	J	106	j	138	è	170	¬	202	╩	234	Ω
11	♂	43	+	75	K	107	k	139	ï	171	½	203	╦	235	δ
12	♀	44	,	76	L	108	l	140	î	172	¼	204	╠	236	∞
13	♪	45	-	77	M	109	m	141	ì	173	¡	205	═	237	φ
14	♫	46	.	78	N	110	n	142	Ä	174	«	206	╬	238	ε
15	☼	47	/	79	O	111	o	143	Å	175	»	207	╧	239	∩
16	►	48	0	80	P	112	p	144	É	176	░	208	╨	240	≡
17	◄	49	1	81	Q	113	q	145	æ	177	▒	209	╤	241	±
18	↕	50	2	82	R	114	r	146	Æ	178	▓	210	╥	242	≥
19	‼	51	3	83	S	115	s	147	ô	179	│	211	╙	243	≤
20	¶	52	4	84	T	116	t	148	ö	180	┤	212	╘	244	⌠
21	§	53	5	85	U	117	u	149	ò	181	╡	213	╒	245	⌡
22	▬	54	6	86	V	118	v	150	û	182	╢	214	╓	246	÷
23	↨	55	7	87	W	119	w	151	ù	183	╖	215	╫	247	≈
24	↑	56	8	88	X	120	x	152	ÿ	184	╕	216	╪	248	°
25	↓	57	9	89	Y	121	y	153	Ö	185	╣	217	┘	249	∙
26	→	58	:	90	Z	122	z	154	Ü	186	║	218	┌	250	·
27	←	59	;	91	[	123	{	155	¢	187	╗	219	█	251	√
28	∟	60	<	92	\	124	¦	156	£	188	╝	220	▄	252	n
29	↔	61	=	93	]	125	}	157	¥	189	╜	221	▌	253	²
30	▲	62	>	94	^	126	~	158	₧	190	╛	222	▐	254	■
31	▼	63	?	95	_	127	⌂	159	ƒ	191	┐	223	▀	255	

Index

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Microsoft Corporation
One Microsoft Way
Redmond, WA 98052-639

Microsoft®

0392 Part No. 28917