WINDOWS API
VOLUME III

WINDOWS 3.1
REFERENCE GUIDE

BORLAND

Windows API Guide

Reference

Volume 3

Version 3.1
for the MS-DOS and PC-DOS
Operating Systems

BORLAND INTERNATIONAL, INC. 1800 GREEN HILLS ROAD
P.O. BOX 660001, SCOTTS VALLEY, CA 95067-0001

PRINTED IN THE USA.
10 9 8 7 6 5 4 3

Contents

Chapter 1 Common dialog box library **1**

Using Color dialog boxes 3
  Color models used by the Color dialog box 4
    RGB color model 4
    HSL color model 6
    Converting HSL values to RGB values 6
  Using the Color dialog box to display basic colors 7
    Initializing the CHOOSECOLOR structure 7
    Calling the ChooseColor function . . 8
  Using the Color dialog box to display custom colors 8
    Initializing the CHOOSECOLOR structure 8
    Calling the ChooseColor function . . 9
Using Font dialog boxes 11
  Displaying the Font dialog box in your application 11
Using Open and Save As dialog boxes . . . 13
  Displaying the Open dialog box in your application 13
  Displaying the Save As dialog box in your application 16
  Monitoring list box controls in an Open or Save As dialog box 18
  Monitoring filenames in an Open or Save As dialog box 19
Using Print and Print Setup dialog boxes 20
  Device drivers and the Print dialog box 21
  Displaying a Print dialog box for the default printer 21
Using Find and Replace dialog boxes 23
  Displaying the Find dialog box 23
  Displaying the Replace dialog box 25
  Processing dialog box messages for a Find or Replace dialog box 26
Customizing common dialog boxes 27
  Appropriate and inappropriate customizations 27
  Hook functions and custom dialog box templates 28
    Hook function 28
    Customizing a dialog box template 31
  Displaying the custom dialog box 32
Supporting help for the common dialog boxes 34
Error detection 35

Chapter 2 Dynamic Data Exchange Management Library **37**

Basic concepts 38
  Client and server interaction 39
  Transactions and the DDE callback function 39
  Service names, topic names, and item names 40
  System topic 40
Initialization 42

Callback function 43
String management 45
Name service 47
Service-name registration 47
Service-name filter 48
Conversation management 48
Single conversations 48
Multiple conversations 52
Data management 54
Transaction management 57
Request transaction 57
Poke transaction 58
Advise transaction 59
Execute transaction 60
Synchronous and asynchronous transactions 61
Transaction control 62
Transaction classes 63
Transaction summary 64
Error detection 66
Monitoring applications 66

Chapter 3 Object linking and embedding libraries 71

Basics of object linking and embedding .. 71
Compound documents 72
Linked and embedded objects 73
Packages 74
Verbs 74
Benefits of object linking and embedding 75
Choosing between OLE and the DDEML 76
Using OLE for standard DDE operations 77
Using both OLE and the DDEML 79
Data transfer in object linking and embedding 79
Client applications 80
Server applications 80
Object handlers 80
Communication between OLE libraries 81
Clipboard conventions 81
Registration 85
Registration database 85
Version control for servers 87
Client user interface 88
New and changed commands 88
Using packages 91
Server user interface 92
Updating objects from multiple-instance servers 92
Updating objects from single-instance servers 93
Object storage formats 93
Client applications 95
Starting a client application 96
Opening a compound document 97
Document management 98
Saving a document 99
Closing a document 99
Asynchronous operations 99
Displaying and printing objects 102
Opening and closing objects 102
Deleting objects 103
Client Cut and Copy commands 103
Creating objects 105
Object-creation functions 105
Paste and Paste Link commands . 107
Undo command 108
Class Name Object command 109
Links command 109
Closing a client application 110
Server applications 111
Starting a server application 112
Opening a document or object 114

Server Cut and Copy commands 115
Update, Save As, and New commands 116
Closing a server application 117
Object handlers 119
Implementing object handlers 119
Creating objects in an object handler . 122
DefCreateFromClip and DllCreateFromClip 122
DefLoadFromStream and DllLoadFromStream 123
Direct use of Dynamic Data Exchange .. 124
Client applications and direct use of Dynamic Data Exchange 124
Server applications and direct use of Dynamic Data Exchange 127
Conversations 128
Items for the system topic 128
Standard item names and notification control 129
Standard commands in DDE execute strings 131
International execute commands 131
Required commands 132
Variants on required commands 134

Chapter 4 Functions 135

AbortDoc 135
AbortProc 136
AllocDiskSpace 136
AllocFileHandles 137
AllocGDIMem 138
AllocMem 139
AllocUserMem 139
CallNextHookEx 140
CallWndProc 140
CBTProc 141
ChooseColor 145
ChooseFont 147
ClassFirst 148
ClassNext 149
CloseDriver 150
CommDlgExtendedError 151
CopyCursor 154
CopyIcon 155
CopyLZFile 155
CPlApplet 157
CreateScalableFontResource 157
DdeAbandonTransaction 160
DdeAccessData 162
DdeAddData 163
DdeCallback 165
DdeClientTransaction 167
DdeCmpStringHandles 170
DdeConnect 172
DdeConnectList 174
DdeCreateDataHandle 176
DdeCreateStringHandle 179
DdeDisconnect 181
DdeDisconnectList 181
DdeEnableCallback 182
DdeFreeDataHandle 183
DdeFreeStringHandle 185
DdeGetData 186
DdeGetLastError 187
DdeInitialize 190
DdeKeepStringHandle 194
DdeNameService 195
DdePostAdvise 197
DdeQueryConvInfo 199
DdeQueryNextServer 200
DdeQueryString 202
DdeReconnect 203
DdeSetUserHandle 204
DdeUnaccessData 205
DdeUninitialize 206

DebugOutput 207
DebugProc 208
DefDriverProc 209
DirectedYield 210
DlgDirSelectComboBoxEx 211
DlgDirSelectEx 212
DragAcceptFiles 213
DragFinish 213
DragQueryFile 214
DragQueryPoint 214
DriverProc 215
EnableCommNotification 217
EnableScrollBar 218
EndDoc 220
EndPage 220
EnumFontFamilies 221
EnumFontFamProc 222
EnumFontsProc 225
EnumMetaFileProc 227
EnumObjectsProc 228
EnumPropFixedProc 230
EnumPropMovableProc 231
EnumTaskWndProc 232
EnumWindowsProc 232
ExitWindowsExec 233
ExtractIcon 234
FindExecutable 234
FindText 236
FMExtensionProc 238
FreeAllGDIMem 239
FreeAllMem 240
FreeAllUserMem 240
GetAspectRatioFilterEx 240
GetBitmapDimensionEx 241
GetBoundsRect 241
GetBrushOrgEx 243
GetCharABCWidths 243
GetClipCursor 244
GetCurrentPositionEx 245
GetCursor 245
GetDCEx 246
GetDriverInfo 247
GetDriverModuleHandle 248
GetExpandedName 249
GetFileResource 250
GetFileResourceSize 251
GetFileTitle 252
GetFileVersionInfo 253
GetFileVersionInfoSize 254
GetFontData 254
GetFreeFileHandles 257
GetFreeSystemResources 257
GetGlyphOutline 258
GetKerningPairs 260
GetMessageExtraInfo 261
GetMsgProc 261
GetNextDriver 262
GetOpenClipboardWindow 263
GetOpenFileName 264
GetOutlineTextMetrics 266
GetQueueStatus 268
GetRasterizerCaps 269
GetSaveFileName 270
GetSelectorBase 272
GetSelectorLimit 273
GetSystemDebugState 273
GetSystemDir 274
GetTextExtentPoint 275
GetTimerResolution 276
GetViewportExtEx 276
GetViewportOrgEx 276
GetWinDebugInfo 277
GetWindowExtEx 278
GetWindowOrgEx 278
GetWindowPlacement 278
GetWindowsDir 279

GetWinMem32Version 280
Global16PointerAlloc 281
Global16PointerFree 282
Global32Alloc 283
Global32CodeAlias 285
Global32CodeAliasFree 286
Global32Free 287
Global32Realloc 287
GlobalEntryHandle 289
GlobalEntryModule 290
GlobalFirst 291
GlobalHandleToSel 292
GlobalInfo 293
GlobalNext 293
GrayStringProc 295
HardwareProc 295
hardware_event 296
hmemcpy 297
_hread 298
_hwrite 298
InterruptRegister 299
Low-stack Faults 301
InterruptUnRegister 302
IsBadCodePtr 303
IsBadHugeReadPtr 304
IsBadHugeWritePtr 304
IsBadReadPtr 305
IsBadStringPtr 305
IsBadWritePtr 306
IsGDIObject 306
IsMenu 307
IsTask 307
JournalPlaybackProc 307
JournalRecordProc 309
KeyboardProc 310
LibMain 311
LineDDAProc 312
LoadProc 313
LocalFirst 314
LocalInfo 315
LocalNext 315
LockInput 316
LockWindowUpdate 317
LogError 318
LogParamError 319
LZClose 322
LZCopy 323
LZDone 324
LZInit 325
LZOpenFile 327
LZRead 329
LZSeek 331
LZStart 332
MapWindowPoints 333
MemManInfo 334
MemoryRead 335
MemoryWrite 336
MessageProc 337
ModuleFindHandle 338
ModuleFindName 339
ModuleFirst 340
ModuleNext 341
MouseProc 342
MoveToEx 343
NotifyProc 343
NotifyRegister 344
NotifyUnRegister 347
OffsetViewportOrgEx 347
OffsetWindowOrgEx 348
OleActivate 349
OleBlockServer 350
OleClone 351
OleClose 352
OleCopyFromLink 352
OleCopyToClipboard 353
OleCreate 354

OleCreateFromClip 355
OleCreateFromFile 357
OleCreateFromTemplate 360
OleCreateInvisible 362
OleCreateLinkFromClip 364
OleCreateLinkFromFile 366
OleDelete 368
OleDraw 369
OleEnumFormats 370
OleEnumObjects 371
OleEqual 372
OleExecute 372
OleGetData 373
OleGetLinkUpdateOptions 374
OleIsDcMeta 375
OleLoadFromStream 376
OleLockServer 377
OleObjectConvert 378
OleQueryBounds 379
OleQueryClientVersion 380
OleQueryCreateFromClip 380
OleQueryLinkFromClip 382
OleQueryName 383
OleQueryOpen 384
OleQueryOutOfDate 384
OleQueryProtocol 385
OleQueryReleaseError 386
OleQueryReleaseMethod 386
OleQueryReleaseStatus 388
OleQueryServerVersion 388
OleQuerySize 389
OleQueryType 389
OleReconnect 390
OleRegisterClientDoc 390
OleRegisterServer 391
OleRegisterServerDoc 393
OleRelease 394
OleRename 394
OleRenameClientDoc 395
OleRenameServerDoc 396
OleRequestData 396
OleRevertClientDoc 397
OleRevertServerDoc 398
OleRevokeClientDoc 399
OleRevokeObject 399
OleRevokeServer 400
OleRevokeServerDoc 400
OleSavedClientDoc 401
OleSavedServerDoc 402
OleSaveToStream 402
OleSetBounds 403
OleSetColorScheme 404
OleSetData 405
OleSetHostNames 406
OleSetLinkUpdateOptions 407
OleSetTargetDevice 408
OleUnblockServer 409
OleUnlockServer 410
OleUpdate 411
OpenDriver 411
PrintDlg 412
QueryAbort 415
QuerySendMessage 415
RedrawWindow 416
RegCloseKey 419
RegCreateKey 420
RegDeleteKey 422
RegEnumKey 423
RegOpenKey 424
RegQueryValue 425
RegSetValue 426
ReplaceText 427
ResetDC 430
ScaleViewportExtEx 431
ScaleWindowExtEx 432
ScrollWindowEx 432

SendDriverMessage 435
SetAbortProc 435
SetBitmapDimensionEx 436
SetBoundsRect 437
SetMetaFileBitsBetter 438
SetSelectorBase 439
SetSelectorLimit 439
SetViewportExtEx 439
SetViewportOrgEx 440
SetWinDebugInfo 442
SetWindowExtEx 443
SetWindowOrgEx 444
SetWindowPlacement 444
SetWindowsHookEx 445
ShellExecute 448
ShellProc 451
SpoolFile 452
StackTraceCSIPFirst 453
StackTraceFirst 454
StackTraceNext 455
StartDoc 456
StartPage 457
SubtractRect 457
SysMsgProc 458
SystemHeapInfo 459
SystemParametersInfo 460
TaskFindHandle 466
TaskFirst 467
TaskGetCSIP 468
TaskNext 468
TaskSetCSIP 469
TaskSwitch 470
TerminateApp 470
TimerCount 471
TimerProc 472
UnAllocDiskSpace 473
UnAllocFileHandles 473
UndeleteFile 474
UnhookWindowsHookEx 474
VerFindFile 475
VerInstallFile 478
VerLanguageName 482
VerQueryValue 484
WindowProc 487
WNetAddConnection 488
WNetCancelConnection 489
WNetGetConnection 489
WordBreakProc 490

Chapter 5 Data types 493

Chapter 6 Messages 503

CB_ADDSTRING 503
CB_DELETESTRING 504
CB_FINDSTRINGEXACT 505
CB_GETDROPPEDCONTROLRECT 505
CB_GETDROPPEDSTATE 506
CB_GETEXTENDEDUI 507
CB_GETITEMHEIGHT 507
CB_SETEXTENDEDUI 508
CB_SETITEMHEIGHT 509
EM_GETFIRSTVISIBLELINE 510
EM_GETPASSWORDCHAR 510
EM_GETWORDBREAKPROC 511
EM_SETREADONLY 511
EM_SETWORDBREAKPROC 512
LB_FINDSTRINGEXACT 513
LB_GETCARETINDEX 514
LB_SETCARETINDEX 514
STM_GETICON 515
STM_SETICON 515
WM_CHOOSEFONT_GETLOGFONT 516
WM_COMMNOTIFY 516
WM_DDE_ACK 517
Posting 519
Receiving 519

WM_DDE_ADVISE 520
Posting 520
Receiving 521
WM_DDE_DATA 521
Posting 522
Receiving 522
WM_DDE_EXECUTE 523
Posting 524
Receiving 524
WM_DDE_INITIATE 524
Sending 525
Receiving 526
WM_DDE_POKE 526
Posting 527
Receiving 527
WM_DDE_REQUEST 527
Posting 528
Receiving 528
WM_DDE_TERMINATE 528
Posting 528
Receiving 529
WM_DDE_UNADVISE 529
Posting 529
Receiving 530
WM_DROPFILES 530
WM_PALETTEISCHANGING 530
WM_POWER 531
WM_QUEUESYNC 532
WM_SYSTEMERROR 532
WM_USER 533
WM_WINDOWPOSCHANGED 534
WM_WINDOWPOSCHANGING ... 534
Notification messages 536
BN_HILITE 536
BN_PAINT 536
BN_UNHILITE 536
CBN_CLOSEUP 537
CBN_SELENDCANCEL 537
CBN_SELENDOK 538
LBN_SELCANCEL 538

Chapter 7 Structures 539
ABC 539
CBT_CREATEWND 540
CBTACTIVATESTRUCT 540
CHOOSECOLOR 541
CHOOSEFONT 544
CLASSENTRY 551
COMSTAT 552
CONVCONTEXT 553
CONVINFO 554
CPLINFO 557
CTLINFO 558
CTLSTYLE 559
CTLTYPE 561
DDEACK 562
DDEADVISE 563
DDEDATA 564
DDEPOKE 565
DEBUGHOOKINFO 566
DEVNAMES 567
DOCINFO 568
DRIVERINFOSTRUCT 569
DRVCONFIGINFO 569
EVENTMSG 570
FINDREPLACE 571
FIXED 575
FMS_GETDRIVEINFO 576
FMS_GETFILESEL 577
FMS_LOAD 578
GLOBALENTRY 579
GLOBALINFO 582
GLYPHMETRICS 583
HARDWAREHOOKSTRUCT 584
HELPWININFO 584
HSZPAIR 585

KERNINGPAIR 586
LOCALENTRY 587
LOCALINFO 590
MAT2 591
MEMMANINFO 592
METAHEADER 593
METARECORD 594
MINMAXINFO 595
MODULEENTRY 596
MONCBSTRUCT 597
MONCONVSTRUCT 598
MONERRSTRUCT 599
MONHSZSTRUCT 600
MONLINKSTRUCT 602
MONMSGSTRUCT 603
MOUSEHOOKSTRUCT 604
NCCALCSIZE_PARAMS 605
NEWCPLINFO 606
NEWTEXTMETRIC 607
NFYLOADSEG 612
NFYLOGERROR 613
NFYLOGPARAMERROR 614
NFYRIP 615
NFYSTARTDLL 616
OLECLIENT 617
OLECLIENTVTBL 617
  Parameters 618
  Return Value 619
  Comments 620
OLEOBJECT 620
OLEOBJECTVTBL 621
  Parameters 624
  Return Value 624
  Comments 624
  Parameters 624
  Return Value 625
  Comments 625
  Parameters 625
  Return Value 625
  Comments 625
  Parameters 626
  Return Value 626
  Parameters 626
  Return Value 626
  Comments 627
  Parameters 627
  Return Value 627
  Comments 627
  See Also 627
  Parameters 627
  Return Value 628
  Parameters 628
  Return Value 628
  Comments 629
OLESERVER 629
OLESERVERDOC 630
OLESERVERDOCVTBL 630
  Parameters 631
  Return Value 631
  Parameters 631
  Return Value 632
  Comments 632
  Parameters 632
  Return Value 633
  Parameters 633
  Return Value 633
  Parameters 633
  Return Value 634
  Comments 634
  Parameters 634
  Return Value 634
  Parameters 635
  Return Value 635
  Comments 635
  Parameters 636
  Return Value 636

Comments 636
OLESERVERVTBL 636
Parameters 637
Return Value 637
Comments 637
Parameters 638
Return Value 638
Comments 638
Parameters 639
Return Value 639
Comments 639
Parameters 640
Return Value 640
Comments 640
Parameters 641
Return Value 641
Comments 641
Parameters 641
Return Value 641
Comments 641
Parameters 642
Return Value 642
Comments 642
OLESTREAM 643
OLESTREAMVTBL 643
Parameters 644
Return Value 644
Comments 644
Parameters 644
Return Value 645
Comments 645
OLETARGETDEVICE 645
OPENFILENAME 646
OUTLINETEXTMETRIC 655
PANOSE 659
POINTFX 664
PRINTDLG 664
RASTERIZER_STATUS 673
SEGINFO 673
SIZE 675
STACKTRACEENTRY 676
SYSHEAPINFO 677
TASKENTRY 678
TIMERINFO 679
TTPOLYCURVE 680
TTPOLYGONHEADER 681
VS_FIXEDFILEINFO 682
WINDEBUGINFO 686
WINDOWPLACEMENT 690
WINDOWPOS 692

Chapter 8 Macros 695

DECLARE_HANDLE 695
DECLARE_HANDLE32 695
FIELDOFFSET 696
GetBValue 696
GetGValue 697
GetRValue 697
MAKELP 697
MAKELPARAM 698
MAKELRESULT 698
OFFSETOF 699
SELECTOROF 699

Chapter 9 Printer escapes 701

MOUSETRAILS 701
POSTSCRIPT_DATA 702
POSTSCRIPT_IGNORE 702
SETALLJUSTVALUES 703

Chapter 10 Dynamic Data Exchange transactions 705

XTYP_ADVDATA 705
XTYP_ADVREQ 706
XTYP_ADVSTART 707
XTYP_ADVSTOP 708
XTYP_CONNECT 708

XTYP_CONNECT_CONFIRM 709
XTYP_DISCONNECT 710
XTYP_ERROR 711
XTYP_EXECUTE 711
XTYP_MONITOR 712
XTYP_POKE 713
XTYP_REGISTER 714
XTYP_REQUEST 715
XTYP_UNREGISTER 715
XTYP_WILDCONNECT 716
XTYP_XACT_COMPLETE 717

Chapter 11 Common dialog box messages 719

COLOROKSTRING 719
FILEOKSTRING 720
FINDMSGSTRING 721
HELPMSGSTRING 721
LBSELCHSTRING 722
SETRGBSTRING 723
SHAREVISTRING 723

Index 725

CHAPTER

1

Common dialog box library

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

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

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

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

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

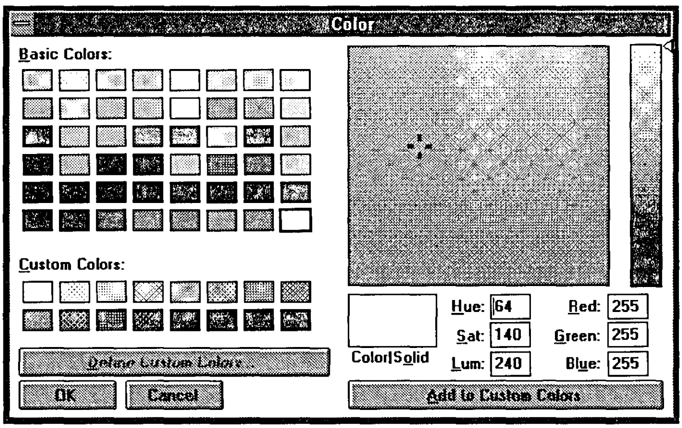
Applications that use the common dialog boxes should specify at least 8K for the stack size, as shown in the following example:

```
NAME cd

EXETYPE WINDOWS

STUB     'WINSTUB.EXE'

CODE     PRELOAD MOVEABLE DISCARDABLE

DATA     PRELOAD MOVEABLE MULTIPLE

HEAPSIZE  1024

STACKSIZE 8192

EXPORTS
  FILEOPENHOOKPROC  @1
```

Using Color dialog boxes

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

Following is a Color dialog box.



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

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

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

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

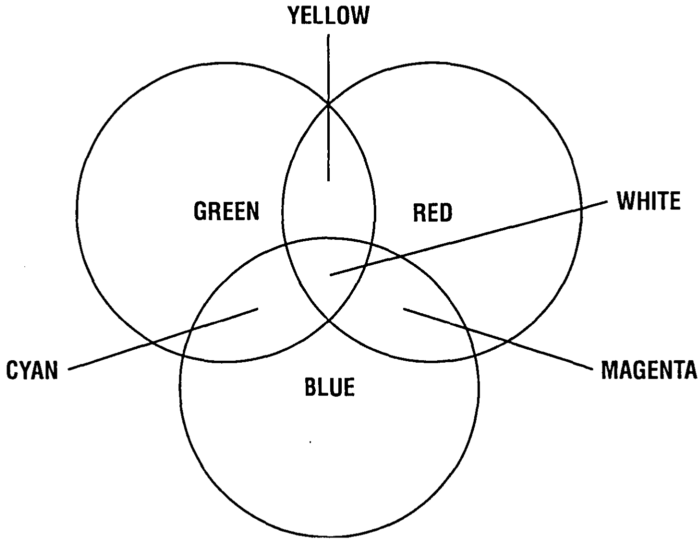
An application can display the Color dialog box in one of two ways: fully open or partially open. When the Color dialog box is displayed partially open, the user cannot change the custom colors.

Color models used by the Color dialog box

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

RGB color model

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



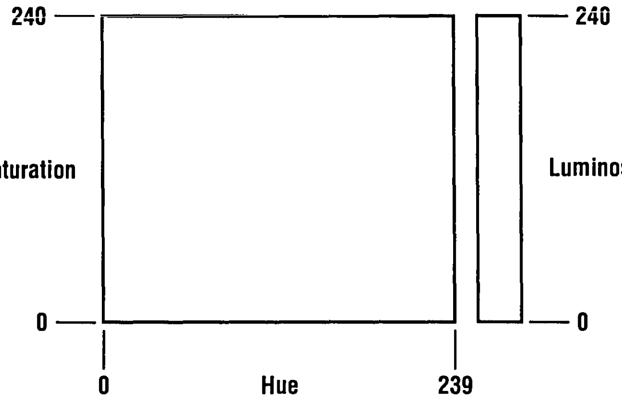
Following are eight colors and their associated RGB values:

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

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

HSL color model

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



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

Converting HSL values to RGB values

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

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

Using the Color dialog box to display basic colors

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

Initializing the CHOOSECOLOR structure

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

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

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

```
/* Color variables */

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

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

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

/* Initialize clr to black. */

clr = RGB(0, 0, 0);

/* Set all structure fields to zero. */

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

/* Initialize the necessary CHOOSECOLOR members. */

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

```
cc.Flags = CC_PREVENTFULLOPEN;

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

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

Calling the ChooseColor function

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

Using the Color dialog box to display custom colors

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

Initializing the CHOOSECOLOR structure

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

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

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

```
/* Color Variables */

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

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

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

Calling the ChooseColor function

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

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

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

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

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

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

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

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

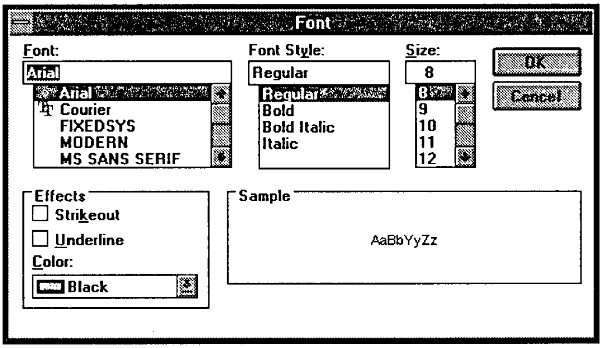
Note that if the Color dialog box is not fully open and the application sends RGB values that do not match one of the basic colors, the system does not update the dialog box. Updates are unnecessary because the spectrum and luminosity controls are not visible when the dialog box is only partially open.

For more information about processing registered window messages, see "Using Find and Replace dialog boxes."

Using Font dialog boxes

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

Following is a Font dialog box.



Displaying the Font dialog box in your application

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

- The attributes of the font that initially is to appear in the dialog box.
- The attributes of the font that the user selected.
- The point size of the font that the user selected.
- Whether the list of fonts corresponds to a printer, a screen, or both.
- Whether the available fonts listed are TrueType only.
- Whether the Effects box should appear in the dialog box.
- Whether dialog box messages should be processed by an application-supplied hook function.

- Whether the point sizes of the selectable fonts should be limited to a specified range.
- Whether the dialog box should display only what-you-see-is-what-you-get (WYSIWIG) fonts. (These fonts are resident on both the screen and the printer.)
- The color that the **ChooseFont** function should use to render text in the Sample box the first time the application displays the dialog box.
- The color that the user selected for text output.

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

1. If the application requires printer fonts, retrieve a device-context handle for the printer and use this handle to set the **hDC** member of the **CHOOSEFONT** structure. (If the Font dialog box displays only screen fonts, this member should be set to NULL.)
2. Set the appropriate flags in the **Flags** member of the **CHOOSEFONT** structure. This setting must include CF_SCREENFONTS, CF_PRINTERFONTS, or CF_BOTH.
3. Set the **rgbColors** member of the **CHOOSEFONT** structure if the default color (black) is not appropriate.
4. Set the **nFontType** member of the **CHOOSEFONT** structure using the appropriate constant.
5. Set the **nSizeMin** and **nSizeMax** members of the **CHOOSEFONT** structure if the CF_LIMITSIZE value is specified in the **Flags** member.
6. Call the **ChooseFont** function.

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

```
LOGFONTlf;
CHOOSEFONTcf;

/* Set all structure fields to zero. */

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

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

```
cf.nFontType = SCREEN_FONTTYPE;

ChooseFont(&cf);
```

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

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

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

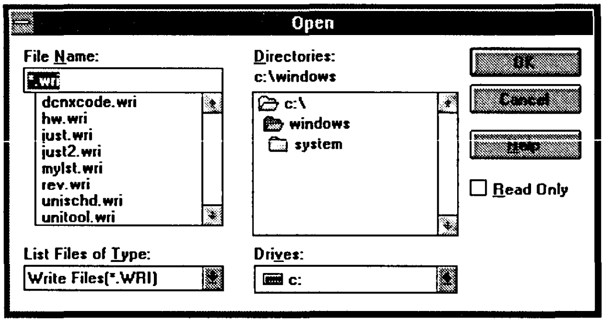
Using Open and Save As dialog boxes

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

Displaying the Open dialog box in your application

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

Following is an Open dialog box.



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

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

1. Store the valid filters in a character array.
2. Set the **lpstrFilter** member to point to this array.
3. Set the **nFilterIndex** member to the value of the index that identifies the default filter.
4. Set the **lpstrFile** member to point to an array that contains the initial filename and receives the selected filename.
5. Set the **nMaxFile** member to the value that specifies the length of the filename array.
6. Set the **lpstrFileTitle** member to point to a buffer that receives the title of the selected file.
7. Set the **nMaxFileTitle** member to specify the length of the buffer.
8. Set the **lpstrInitialDir** member to point to a string that specifies the initial directory. (If this member does not point to a valid string, it must be set to 0 or point to a string that is set to NULL.)

9. Set the **lpstrTitle** member to point to a string specifying the name that should appear in the title bar of the dialog box. (If this pointer is NULL, the title will be Open.)

10. Initialize the **lpstrDefExt** member to point to the default extension. (This extension can be 0, 1, 2, or 3 characters long.)

11. Call the **GetOpenFileName** function.

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

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

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

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

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

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

/* Set all structure members to zero. */

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

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

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

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

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

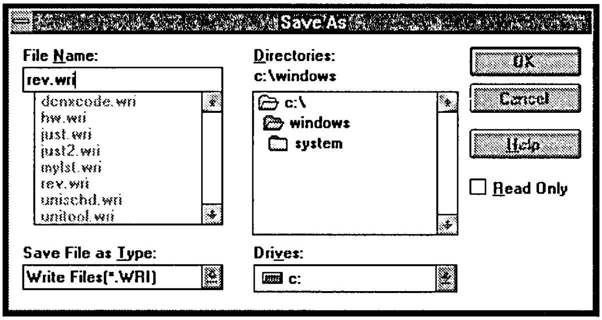
The vertical bars in this string are used as wildcards. After using the **LoadString** function to retrieve the string, the wildcards are replaced with NULL. The wildcard can be any unique character and must be included as the last character in the string. Initializing strings in this manner guarantees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

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

Displaying the Save As dialog box in your application

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

Following is a Save As dialog box.



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

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

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

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

GetSystemDirectory(szDirName, sizeof(szDirName));

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

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

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

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

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

/* Initialize the OPENFILENAME members. */

szFile[0] = '\0';

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

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

The string referred to by the IDS_FILTERSTRING constant in the preceding example is defined in the resource-definition file. It is used in exactly the same way as the IDS_FILTERSTRING constant discussed in "Displaying the Open dialog box in your application."

Monitoring list box controls in an Open or Save As dialog box

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

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

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

For an example that registers a common dialog box message, see "Using Find and Replace dialog boxes."

Monitoring filenames in an Open or Save As dialog box

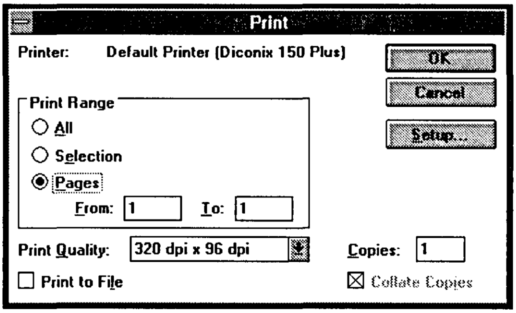
Applications can alter the normal processing of an Open or Save As dialog box by monitoring which filename the user types and by performing other, unique operations. For example, one application could prevent the user from closing the dialog box if the selected filename is prohibited; another application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

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

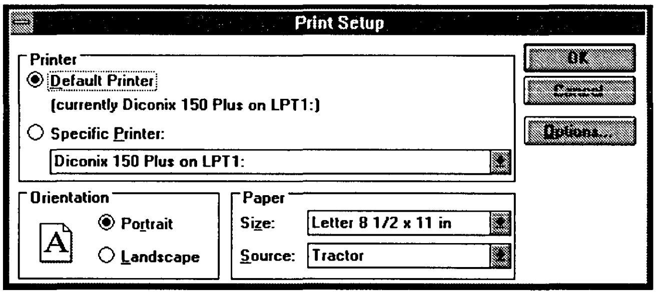
Using Print and Print Setup dialog boxes

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

Following is a Print dialog box.



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



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

Device drivers and the Print dialog box

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

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

Displaying a Print dialog box for the default printer

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

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

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

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

1. Setting the PD_RETURNDC flag in the **Flags** member of the **PRINTDLG** structure. (This flag should only be set if the application requires a device-context handle.)

2. Initializing the **lStructSize, hDevMode,** and **hDevNames** members.

3. Calling the **PrintDlg** function and passing a pointer to the **PRINTDLG** structure just initialized.

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

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

```
PRINTDLGpd;

/* Set all structure members to zero. */

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

/* Initialize the necessary PRINTDLG structure members. */

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

/* Print a test page if successful. */

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

    /* Print text and rectangle. */

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

Using Find and Replace dialog boxes

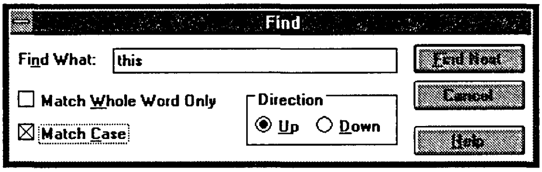
The Find dialog box and the Replace dialog box are similar in appearance. You can use the Find dialog box to add string-search capabilities to your application and use the Replace dialog box to add both string-search and string-substitution capabilities.

Displaying the Find dialog box

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

- The string that the application should find
- Whether the string specifies a complete word or part of a word
- Whether the application should match the case of the specified string
- The direction in which the application should search (preceding or following the current cursor location)
- Whether the application should resume the search, searching for the next occurrence of the string

Following is a Find dialog box.



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

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

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

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

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

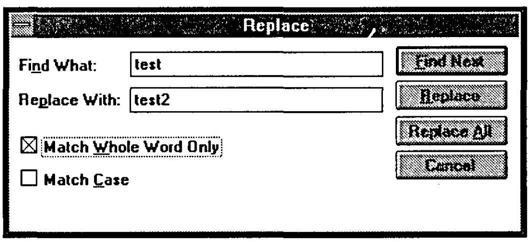
```
FINDREPLACEfr;

/* Set all structure fields to zero. */

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

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

hDlg = FindText(&fr);

break;
```

Displaying the Replace dialog box

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

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

Following is a Replace dialog box.



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

Processing dialog box messages for a Find or Replace dialog box

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

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

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

```
UINT uFindReplaceMsg;

/* Register the FindReplace message. */

uFindReplaceMsg = RegisterWindowMessage(FINDMSGSTRING);
```

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

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

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

Customizing common dialog boxes

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

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

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

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

Appropriate and inappropriate customizations

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

Hook functions and custom dialog box templates

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

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

Hook function

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

```
#defineSTRICT

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

OPENFILENAME ofn;

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

    GetSystemDirectory((LPSTR)szDirName, 255);

    /* Initialize the OPENFILENAME members. */

    szFile[0] = '\0';
    ofn.lStructSize = sizeof(OPENFILENAME);
    ofn.hwndOwner = hwnd;
    ofn.hInstance = hInst;
```

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

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

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

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

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

```
UINTCALLBACKFileOpenHookProc(HWNDhdlg,UINTmsg,WPARAM
     wParam, LPARAM lParam)
{
    switch(msg) {
        case WM_INITDIALOG:
            return TRUE;

        case WM_COMMAND:

            /* Use IsDlgButtonChecked to set lCustData. */

            if (wParam == IDOK) {

                /* Set backup flag. */

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

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

    /* Allow standard processing. */

    return FALSE;
}
```

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

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

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

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

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

```
NAME cd

EXETYPE WINDOWS

STUB    'WINSTUB.EXE'

CODE    PRELOAD MOVEABLE DISCARDABLE

DATA    PRELOAD MOVEABLE MULTIPLE

HEAPSIZE  1024

STACKSIZE8192

EXPORTS
  FILEOPENHOOKPROC  @1
```

Customizing a dialog box template

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

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

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

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

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

END
```

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

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

Displaying the custom dialog box

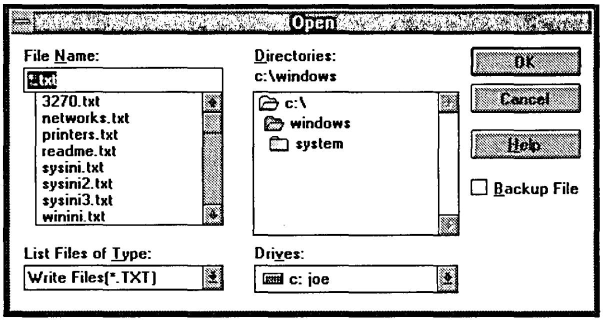
After your application creates the hook function and the dialog box template, it should set the members of the structure for the common dialog box being customized and call the appropriate function to display the custom dialog box.

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

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

if(GetOpenFileName(&ofn)){

    hf = _lopen(ofn.lpstrFile, OF_READWRITE);

    /* Create the backup file. */

    if (ofn.lCustData) {

        /* Process files with extension. */

        if (ofn.nFileExtension){

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

        }/*endif */

        /* Process files without extension. */

        else {

           i=0;

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

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

        pszNewPAFN = lstrcat(szChar, "BAK");

        /* Create the backup file. */

        hfBackup = _lcreat(pszNewPAFN, 0);


        /* Copy contents of original file to the backup file. */

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

    /* File operations begin here. */

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

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



Supporting help for the common dialog boxes

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

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

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

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

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

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

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

```
wParam == pshHelp
```

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

Error detection

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

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

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

FRERR_FINDREPLACECODES	Error codes for the Find and Replace common dialog boxes. These errors are in the range 0x4000 through 0x4FFF.
CCERR_CHOOSECOLORCODES	Error codes for the Color common dialog box. These errors are in the range 0x5000 through 0x5FFF.

C H A P T E R

2

Dynamic Data Exchange Management Library

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

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

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

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

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

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

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

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

Basic concepts

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

Client and server interaction

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

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

Transactions and the DDE callback function

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

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

For a detailed discussion of transactions, see "Transaction management."

Service names, topic names, and item names

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

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

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

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

System topic

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

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

For more information about initiating DDE conversations, see "Conversation management."

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

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

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

Initialization

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

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

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

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

Transaction filters optimize system performance by preventing the DDEML from passing unwanted transactions to the application's DDE callback function. An application sets the transaction filters when it calls the **DdeInitialize** function. An application should specify a transaction filter flag for each type of transaction that it does not process in its callback function. An application can change its transaction filters with a subsequent call to the **DdeInitialize** function.

For more information about transactions, see "Transaction management."

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

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

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

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

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

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

Callback function

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

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

        case XTYP_ADVDATA:
            .
            .
            .
            return (HDDEDATA) DDE_FACK;

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

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

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

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

String management

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

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

```
HSZ hszServName;
HSZ hszSysTopic;

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Name service

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

Service-name registration

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

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

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

Service-name filter

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

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

Conversation management

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

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

Single conversations

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

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

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

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

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

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

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

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

```
#define CTOPICS 5

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

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

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

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

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

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

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

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

```
#define CTOPICS 2

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

if (type == XTYP_WILDCONNECT) {

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

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

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

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

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

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

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

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

Multiple conversations

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Data management

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

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

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

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

TIMEtmTime;
HSZhszTime;
HSZhszNow;
HDDEDATAEXPENTRYDdeProc(wType, wFmt, hConv, hsz1, hsz2,
    hData, dwData1, dwData2)
WORDwType;
WORDwFmt;
HCONVhConv;
HSZhsz1;
HSZhsz2;
HDDEDATAhData;
DWORDdwData1;
DWORDdwData2;
{
    char szBuf[32];

    switch (wType) {

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

                /* Copy formatted time string to buffer. */

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

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

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

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

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

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

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

case XTYP_ADVDATA:

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

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

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

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

Transaction management

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

Request transaction

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

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

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

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

Poke transaction

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

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

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

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

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

Advise transaction

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

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

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

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

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

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

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

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

Execute transaction

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

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

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

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

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

Synchronous and asynchronous transactions

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

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

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

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

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

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

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

Transaction control

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

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

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

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

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

Transaction classes

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

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

Class	Return value	Transaction
XCLASS_BOOL	TRUE or FALSE	XTYP_ADVSTART XTYP_CONNECT
XCLASS_DATA	A data handle, CBR_BLOCK, or NULL	XTYP_ADVREQ XTYP_REQUEST XTYP_WILDCONNECT

Class	Return value	Transaction
XCLASS_FLAGS	A transaction flag: DDE_FACK, DDE_FBUSY, or DDE_FNOTPROCESSED	XTYP_ADVDATA XTYP_EXECUTE XTYP_POKE
XCLASS_NOTIFICATION	None	XTYP_ADVSTOP XTYP_CONNECT_CONFIRM XTYP_DISCONNECT XTYP_ERROR XTYP_REGISTER XTYP_UNREGISTER XTYP_XACT_COMPLETE

Transaction summary

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

Transaction type	Receiver	Cause
XTYP_ADVDATA	Client	A server responded to an XTYP_ADVREQ transaction by returning a data handle.
XTYP_ADVREQ	Server	A server called the **DdePostAdvise** function, indicating that the value of a data item in an advise loop had changed.
XTYP_ADVSTART	Server	A client specified the XTYP_ADVSTART transaction type in a call to the **DdeClient-Transaction** function.
XTYP_ADVSTOP	Server	A client specified the XTYP_ADVSTOP transaction type in a call to the **DdeClient-Transaction** function.
XTYP_CONNECT	Server	A client called the **DdeConnect** function, specifying a service name and topic name supported by the server.

Transaction type	Receiver	Cause
XTYP_CONNECT_CONFIRM	Server	The server returned TRUE in response to an XTYP_CONNECT or XTYP_WILDCONNECT transaction.
XTYP_DISCONNECT	Client/ Server	A partner in a conversation called the **DdeDisconnect** function, causing both partners to receive this transaction.
XTYP_ERROR	Client/ Server	A critical error has occurred. The DDEML may not have sufficient resources to continue.
XTYP_EXECUTE	Server	A client specified the XTYP_EXECUTE transaction type in a call to the **DdeClient-Transaction** function.
XTYP_MONITOR	DDE monitoring application	A DDE event occurred in the system. For more information about DDE monitoring applications, see "Monitoring applications."
XTYP_POKE	Server	A client specified the XTYP_POKE transaction type in a call to the **DdeClientTransaction** function.
XTYP_REGISTER	Client/ Server	A server application used the **DdeName-Service** function to register a service name.
XTYP_REQUEST	Server	A client specified the XTYP_REQUEST transaction type in a call to the **DdeClient-Transaction** function.
XTYP_UNREGISTER	Client/ Server	A server application used the **DdeName-Service** function to unregister a service name.

Transaction type	Receiver	Cause
XTYP_WILDCONNECT	Server	A client called the **DdeConnect** or **DdeConnectList** function, specifying NULL for the service name, the topic name, or both.
XTYP_XACT_COMPLETE	Client	An asynchronous transaction, sent when the client specified the TIMEOUT_ASYNC flag in a call to the **DdeClientTransaction** function, has concluded.

Error detection

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

Monitoring applications

Microsoft Windows DDESpy (DDESPY.EXE) monitors DDE activity in the system. You can use DDESpy as a tool for debugging your DDE applications.

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

To receive notifications from the DDEML, an application must have registered itself as a DDE monitor by specifying the

APPCLASS_MONITOR flag in a call to the **DdeInitialize** function. In this same call, the application can specify one or more monitor flags to indicate the types of events of which the DDEML is to notify the application's callback function. The following table describes each of the monitor flags an application can specify:

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

The following example shows how to register a DDE monitoring application so that its DDE callback function receives notifications of all DDE events:

```
DWORD idInst;
PFNCALLBACK lpDdeProc;
hInst = hInstance;

lpDdeProc = (PFNCALLBACK) MakeProcInstance(
  (FARPROC) DDECallback, /* points to callback function      */
  hInstance);           /* instance handle                 */

if(DdeInitialize(
      (LPDWORD) &idInst,  /* instance identifier             */
      lpDdeProc,          /* points to callback function      */
      APPCLASS_MONITOR |  /* this is a monitoring application */
      MF_CALLBACKS     |  /* monitor callback functions      */
      MF_CONV          |  /* monitor conversation data       */
      MF_ERRORS        |  /* monitor DDEML errors            */
      MF_HSZ_INFO      |  /* monitor data-handle activity    */
      MF_LINKS         |  /* monitor advise loops            */
      MF_POSTMSGS      |  /* monitor posted DDE messages     */
      MF_SENDMSGS,        /* monitor sent DDE messages       */
      0L))                /* reserved                        */
    return FALSE;
```

The DDEML informs a monitoring application of a DDE event by sending an XTYP_MONITOR transaction to the application's DDE callback function. During this transaction, the DDEML passes a monitor flag that specifies the type of DDE event that has occurred and a handle of a global memory object that contains detailed information about the event. The DDEML provides a set of structures that the application can use to extract the information from the memory object. There is a corresponding structure for each type of DDE event. The following table describes each of these structures.

Structure	Description
MONCBSTRUCT	Contains information about a transaction.
MONCONVSTRUCT	Contains information about a conversation.
MONERRSTRUCT	Contains information about the latest DDE error.
MONLINKSTRUCT	Contains information about an advise loop.
MONHSZSTRUCT	Contains information about a string handle.
MONMSGSTRUCT	Contains information about a DDE message that was sent or posted.

The following example shows the DDE callback function of a DDE monitoring application that formats information about each string handle event and then displays the information in a window. The function uses the **MONHSZSTRUCT** structure to extract the information from the global memory object.

```
HDDEDATA CALLBACK DDECallback(wType, wFmt, hConv, hsz1, hsz2,
    hData, dwData1, dwData2)
WORD wType;
WORD wFmt;
HCONV hConv;
HSZ hsz1;
HSZ hsz2;
HDDEDATA hData;
DWORD dwData1;
DWORD dwData2;
{
    LPVOID lpData;
    char *szAction;
    char buf[256];
    DWORD cb;

    switch (wType) {
        case XTYP_MONITOR:

            /* Obtain a pointer of the global memory object. */

            if (lpData = DdeAccessData(hData, &cb)) {

                /* Examine the monitor flag. */
```

```
            switch (dwData2) {
                case MF_HSZ_INFO:

#definePHSZS((MONHSZSTRUCTFAR*)lpData)

                    /*
                     * The global memory object contains
                     * string-handle data. Use the MONHSZSTRUCT
                     * structure to access the data.
                     */

                    switch (PHSZS->fsAction) {

                        /*
                         * Examine the action flags to determine
                         * the action performed on the handle.
                         */

                        case MH_CREATE:
                            szAction = "Created";
                            break;

                        case MH_KEEP:
                            szAction = "Incremented";
                            break;

                        case MH_DELETE:
                            szAction = "Deleted";
                            break;

                        case MH_CLEANUP:
                            szAction = "Cleaned up";
                            break;

                        default:
                            DdeUnaccessData(hData);
                            return ((HDDEDATA) 0);
                    }

                     /* Write formatted output to a buffer. */

                     wsprintf(buf,
                        "Handle %s, Task: %x, Hsz: %lx(%s)",
                        (LPSTR) szAction, PHSZS->hTask, PHSZS->hsz,
                        (LPSTR) PHSZS->str);
                     .
                     . /* Display text in window or write to file. */
                     .

                     break;

#undefPHSZS

                  .
                  . /* Process other MF_* flags. */
                  .

                  default:
                      break;
```

```
                    }
                }

                /* Free the global memory object. */

                DdeUnaccessData(hData);
                break;

            default:
                break;
        }
        return ((HDDEDATA) 0);
}
```

C H A P T E R

3

Object linking and embedding libraries

This chapter describes the implementation of object linking and embedding (OLE) for applications that run with the Microsoft Windows operating system. The chapter also describes how an application can use linked and embedded objects to create compound documents. The following topics are related to the information in this chapter:

- Dynamic data exchange (DDE)
- Clipboard
- Registration database
- Dynamic-link libraries
- Multiple document interface

This chapter does not go into detail about the recommended user interface for applications that use linked and embedded objects.

Basics of object linking and embedding

This section explains some basic OLE concepts and compares OLE functionality to that of the Dynamic Data Exchange Management Library (DDEML).

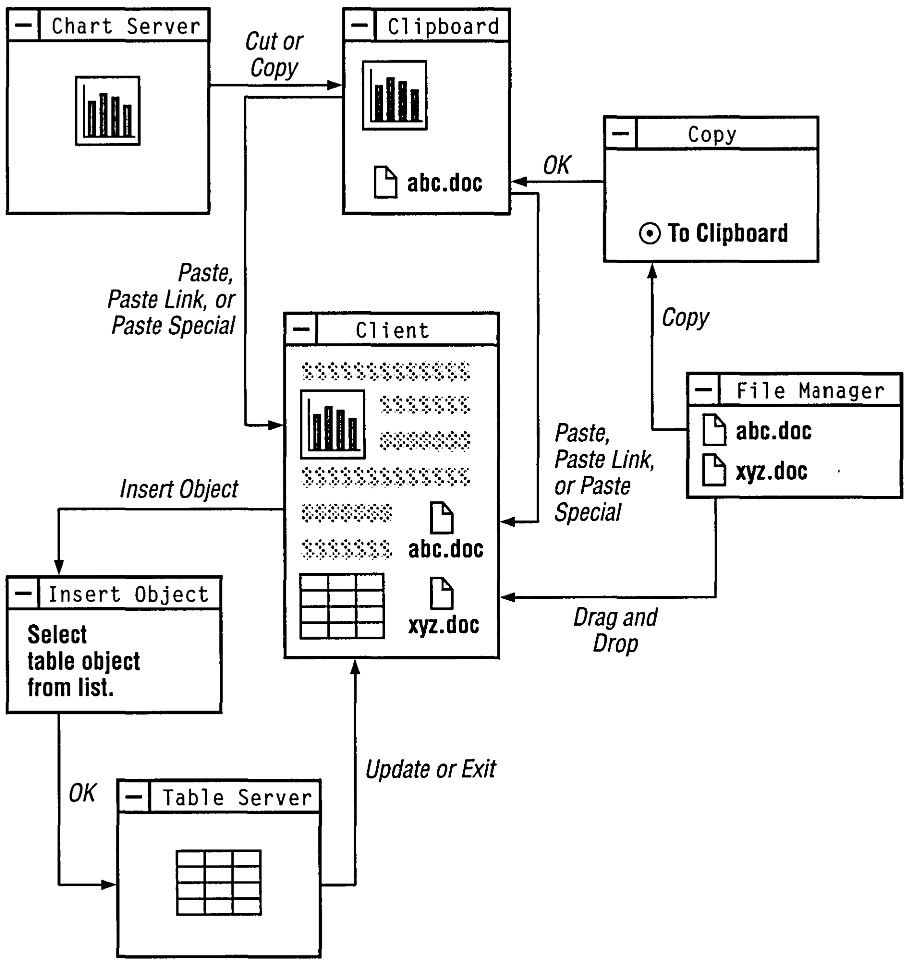
Compound documents

An application that uses OLE can cooperate with other OLE applications to produce a document containing different kinds of data, all of which can be easily manipulated by the user. The user editing such a document is able to improve the document by using the best features of many different applications. An application that implements OLE gives its users the ability to move away from an application-centered view of computing and toward a document-centered view. In application-centered computing, the tool used to complete a task is often a single application; whereas, in document-centered computing, a user can combine the advantages of many tools to complete a job.

A document that uses linked and embedded objects can contain many kinds of data in many different formats; such a document is called a compound document. A compound document uses the facilities of different OLE applications to manipulate the different kinds of data it displays. Any kind of data format can be incorporated into a compound document; with little or no extra code, OLE applications can even support data formats that have not yet been invented. The user working with a compound document does not need to know which data formats are compatible with one another or how to find and start any applications that created the data. Whenever a user chooses to work with part of a compound document, the application responsible for that part of the document starts automatically.

For example, a compound document could be a brochure that included text, charts, ranges of cells in a spreadsheet, and illustrations. The information could be embedded in the document, or the document could contain links to certain information instead of containing the information itself. The user working with the brochure could automatically switch between the applications that produced its components.

The following illustration shows the relationships between a compound document and its linked and embedded objects.



Linked and embedded objects

An object is any data that can be presented in a compound document and manipulated by a user. Anything from a single cell in a spreadsheet to an entire document can be an object. When an object is incorporated into a document, it maintains an association with the application that produced it. That association can be a link, or the object can be embedded in the file.

If the object is linked, the document provides only minimal storage for the data to which the object is linked, and the object can be updated automatically whenever the data in the original application changes. For example, if a range of spreadsheet cells were linked to information in a text file, the data would be stored in some other file and only a link to the data would be saved with the text file.

If an object is embedded, all the data associated with it is saved as part of the file in which it is embedded. If a range of spreadsheet cells were embedded in a text file, the data in the cells would be saved with the text file, including any necessary formulas; the name of the server for the spreadsheet cells would be saved along with this data. The user could select this embedded object while working with the text file, and the spreadsheet application would be started automatically for editing those cells. The presentation and the behavior of the data is the same for a linked and an embedded object.

Packages

A package is a type of OLE object that encapsulates another object, a file, or a command line inside a graphic representation (such as an icon or bitmap). When the user double-clicks the graphic object, the OLE libraries activate the object inside the package. The package itself is always an embedded object, not a link. The contents of a package can be an embedded object, a link, or even a file dropped from Windows File Manager.

Packages are useful for presenting compact token views of large files or OLE objects. An application could also use a package as it would use a hyperlink—that is, to connect information in different documents.

Windows version 3.1 includes the application Microsoft Windows Object Packager (PACKAGER.EXE). With Packager, a user can associate a file or data selection with an icon or graphic.

Verbs

The types of actions a user can perform on an object are called verbs. Two typical verbs for an object are Play and Edit.

The nature of an object determines its behavior when a user works with it. The most typical use for some objects, such as voice annotations and animated scripts, is to play them. For example, a user could play an animated script by double-clicking it. In this case, Play is the primary verb for the object.

For other objects, the most typical use is to edit them. In the case of text produced by a word processor, for example, the primary verb could be Edit.

The client application typically specifies the primary verb when the user double-clicks an object. However, the server application determines the meaning of that verb. A user can invoke an object's subsidiary verbs by using the *Class Name* Object command or the Links dialog box. For more information about these topics, see "Client user interface."

The action taken when a user double-clicks a package is that of the primary verb of the object inside the package. The secondary verb for a packaged object is Edit Package; when the user chooses this verb, Packager starts. The user can use Packager to gain access to the secondary verb for the object inside the package.

Many objects support only one verb—for example, an object created by a text editor might support only Edit. If an object supports only one verb, that verb is used no matter what the client application specifies. For more information about verbs, see "Registration."

Benefits of object linking and embedding

OLE offers the following benefits:

- An application can specialize in performing one job very well. For example, a drawing application that implements OLE does not need any text-editing tools; a user could put text into the drawing and edit that text by using any text editor that supports OLE.
- An application is automatically extensible for future data formats, because the content of an object does not matter to the containing document.
- A user can concentrate on the task instead of on any software required to complete the task.
- A file can be more compact, because linking to objects allows a file to use an object without having to store that object's data.
- A document can be printed or transmitted without using the application that originally produced the document.
- Linked objects in a file can be updated dynamically.

Future implementations of this protocol could take advantage of a wide variety of object types. For example, the user of a voice-recorder application could dictate a comment, package the comment as an object with a visual representation, and embed the graphic as an object in a text file. When a user double-clicked the graphic for this object (a pair of lips, perhaps), the voice-recorder application would play the recorded comment. Linked and embedded objects also lend themselves to implementations such as animated drawings, executable macro scripts, hypertext, and annotations.

Choosing between OLE and the DDEML

Applications can exchange data by using either OLE or the DDEML. Unless an application has a strong requirement for managing multiple items in a single conversation with another application, the application should use OLE instead of the DDEML.

Both OLE and the DDEML are message-based systems supported by dynamic-link libraries. Developers are encouraged to use these libraries rather than using the underlying message-based protocols. For more information about the message-based OLE protocol, see "Direct use of Dynamic Data Exchange."

Unlike OLE, the DDEML supports multiple items per conversation. With OLE, a client needing links to several objects in a document must establish a separate conversation for each object.

OLE offers the following advantages that the DDEML does not:

Advantage	Description
Extensibility to future enhancements	The OLE libraries may be updated in future releases to support new data formats, link tracking, editing without exiting the client application, and other enhancements that will not be immediately available to applications that use the DDEML.
Persistent embedding and linking of objects	The OLE libraries do most of the work of activating objects when an embedded document is reopened, by reestablishing the conversation between a client and server. In contrast, establishing a DDE link (DDE advise loop) is the responsibility of either the user (if the link is not persistent) or of the application (if the link is persistent).

Advantage	Description
Rendering of common data formats	The OLE libraries assume the burden of rendering common data formats on a display context. DDE applications, however, must do this work themselves.
Server rendering of specialized data formats	The OLE libraries facilitate the rendering of specialized data formats in the client's display context. (The server application or object handler actually performs the rendering.) The client application has to do very little work to render the embedded or linked data in its display context. Such rendering of embedded or linked data is beyond the scope of the DDEML alone.
Activating embedded and linked objects	The OLE libraries support activating a server to edit a linked or embedded object or to render data. Activating servers for data rendering and editing is beyond the scope of the DDEML.
Creating objects and links from the clipboard	The OLE libraries do most of the work when an application is using the clipboard to copy and paste links or exchange objects. In contrast, DDE applications must call the Windows clipboard functions directly to perform such operations.
Creating objects and links from files	The OLE libraries provide direct support for using files to exchange data. No DDE protocol is defined for this purpose.

The OLE libraries use DDE messages instead of the DDEML, because the libraries were written before the DDEML was available.

Using OLE for standard DDE operations

Although most of the OLE application programming interface (API) was designed for linked and embedded objects, it can also be applied to standard DDE items. In particular, an application can use the OLE API to perform the following DDE tasks:

- Initializing conversations based on application and topic names or wildcards.
- Requesting data for named items in negotiated formats from a server.
- Establishing an advise loop—that is, requesting that a DDE server notify the client of changes to the values of specified

items and, optionally, that the server send the data when the change occurs.

- Sending data from a server to a client.
- Poking data from a client to a server.
- Sending a DDE command. (This is supported by the **OleExecute** function.)

An OLE client application receives a pointer to an **OLEOBJECT** structure; this structure includes class name, document name, and item name information. These names correspond exactly to DDE counterparts, as follows:

OLE name	DDE name
Class name	Service name (formerly called "application name")
Document name	Topic name
Item name	Item name

The client can use the **OleCreateFromFile** function to make an object and specify all three names. If the client application needs multiple items from the same topic, it must have an **OLEOBJECT** structure for each item, which causes a DDE conversation to be created for each item.

The client library maps OLE functions that work on the **OLEOBJECT** structure to DDE messages as follows:

OLE function	DDE message
OleExecute	WM_DDE_EXECUTE
OleRequestData	WM_DDE_REQUEST
OleSetData	WM_DDE_POKE

Some functions (such as **OleActivate**) are too complicated for this one-to-one mapping of function to DDE message. For these functions, the DDE message depends on the circumstance.

If a client application needs to duplicate the functionality of WM_DDE_ADVISE with OLE, the client must create the link with **olerender_format** for the *renderopt* parameter, specify the required format, and use the **OleGetData** function to retrieve the value when the callback function receives the OLE_CHANGED notification. If more than one item or format is required, the client must create an **OLEOBJECT** structure for each item/format pair. Although this method creates a conversation for each advise transaction, it may be inefficient if the client needs to create many such conversations.

A server application can make itself accessible to DDE by calling the **OleRegisterServer** function to make the System topic available and the **OleRegisterServerDoc** function to make other topics available. When a client connects and asks for an item, the server library calls the **GetObject** function in the server's **OLESERVERDOCVTBL** structure, followed by other server-implemented functions that are appropriate to the client's request. (Usually, the library calls the **GetData** function in the server's **OLEOBJECTVTBL** structure.) As long as the object allocated by the call to **GetObject** has not been released, the server should send a notification when the item has changed, so that the OLE libraries can send data to clients that have sent WM_DDE_ADVISE.

Using both OLE and the DDEML

Some applications may need features supported only by OLE and may also need to use the DDEML to support simultaneous links for many items that are updated frequently. Client applications of this kind can use the OLE libraries to initiate conversations with OLE servers and the DDEML to initiate conversations with DDE servers.

Server applications that need to support both OLE and the DDEML must use different service names (DDE application names) for OLE and DDE conversations; otherwise, the OLE and DDEML libraries cannot determine which library should respond when an initiation request is received. Typically, the application changes the service name for the OLE conversation in this case, because other applications and the user must use the service name for the DDE conversation, but the OLE service name is hidden.

Data transfer in object linking and embedding

This section gives a brief overview of how applications share information under OLE. Details of the implementation are given in later sections of this chapter.

Applications use three dynamic-link libraries (DLLs), OLECLI.DLL, OLESVR.DLL, and SHELL.DLL, to implement object linking and embedding. Object linking and embedding is supported by OLECLI.DLL and OLESVR.DLL. The registration database is supported by SHELL.DLL.

Client applications

An OLE client application can accept, display, and store OLE objects. The objects themselves can contain any kind of data. A client application typically identifies an object by using a distinctive border or other visual cue.

Server applications

An OLE server is any application that can edit an object when the OLE libraries inform it that the user of a client application has selected the object. (Some servers can perform operations on an object other than editing.) When the user double-clicks an object in a client application, the server associated with that object starts and the user works with the object inside the server application. When the server starts, its window is typically sized so that only the object is visible. If the user double-clicks a linked object, the entire linked file is loaded and the linked portion of the file is selected. For embedded objects, the user chooses the Update command from the File menu to save changes to the object and chooses Exit when finished.

Many applications are capable of acting as both clients and servers for linked and embedded objects.

Object handlers

Some OLE server applications implement an additional kind of OLE library called an object handler. Object handlers are dynamic-link libraries that act as intermediaries between client and server applications. Typically, an object handler is supplied by the developers of a server application as a way of improving performance. For example, an object handler could be used to redraw a changed object if the presentation data for that object could not be rendered by the client library.

Communication between OLE libraries

Client applications use functions from the OLE API to inform the client library, OLECLI.DLL, that a user wants to perform an operation on an object. The client library uses DDE messages to communicate with the server library, OLESVR.DLL. The server library is responsible for starting and stopping the server application, directing the interaction with the server's callback functions, and maintaining communication with the client library.

When a server application modifies an embedded object, the server notifies the server library of changes. The server library then notifies the client library, and the client library calls back to the client application, informing it that the changes have occurred. Typically, the client application then forces a repaint of the embedded object in the document file. If the server changes a linked object, the server library notifies the client library that the object has changed and should be redrawn.

Clipboard conventions

When first embedding or linking an object, OLE client and server applications typically exchange data by using the clipboard. When a server application puts an object on the clipboard, it represents the object with data formats, such as Native data, OwnerLink data, ObjectLink data, and a presentation format. The order in which these formats are put on the clipboard is very important, because the order determines the type of object. For example, if the first format is Native and the second is OwnerLink, client applications can use the data to create an embedded object. If the first format is OwnerLink, however, the data describes a linked object.

Native data completely defines an object for a particular server. The data can be meaningful only to the server application. The client application provides storage for Native data, in the case of embedded objects.

OwnerLink data identifies the owner of a linked or embedded object.

Presentation formats allow the client library to display the object in a document. CF_METAFILEPICT, CF_DIB, and CF_BITMAP are typical presentation formats. Native data can be used as a presentation format, typically when an object handler has been defined for that class of data. Native data cannot be used twice in the definition of an object, however; if the server puts Native and OwnerLink data on the clipboard to describe an embedded object, it cannot use Native data as a presentation format for that object. The ability of object handlers to use Native data as the presentation data accounts for the significance of the order of the formats: the order is the only way to distinguish between an embedded object and a link that uses Native data for its presentation.

ObjectLink data identifies a linked object's class and document and the item that is the source for the linked object. (If the item name specified in the ObjectLink format is NULL, the link refers to the entire server document.)

The following table describes the contents of the ObjectLink, OwnerLink, and Native clipboard formats:

Format name	Contents
ObjectLink	Null-terminated string for class name, null-terminated string for document name, string for item name with two terminating null characters.
OwnerLink	Null-terminated string for class name, null-terminated string for document name, string for item name with two terminating null characters.
Native	Stream of bytes interpreted only by the server application or object-handler library. This format can be unique to the server application and must allow the server to load and work with the object.

Although the ObjectLink and OwnerLink formats contain the same information, the OLE libraries use them differently. The libraries use OwnerLink format to identify the owner of an object (which can be different from the source of the object) and ObjectLink format to identify the source of the data for an object.

The class name in the ObjectLink or OwnerLink format is a unique name for a class of objects that a server supports. Server applications register the class name or names they support in the registration database. (For example, the class name used by Windows Paintbrush™ is PBrush.) An application can use the class name to look up information about a server in the registration database. (For more information about registration, see "Registration.") The document name is typically a fully qualified path that identifies the file containing a document. The item name uniquely identifies the part of a document that is defined as an object. Item names are assigned by server applications; an item name can be any string that the server uses to identify part of a document. Items names cannot contain the forward-slash (/) character.

Data in OwnerLink or ObjectLink format could look like the following example:

```
MicrosoftExcel
Worksheet\0c:\directry\docname.xls\0R1C1:R5C3\0\0
```

The order in which various data formats are put on the clipboard depends on the type of data being copied to the clipboard and the capabilities of the server application. The following table shows the order of clipboard data formats for four different types of data selections. An object does not necessarily use all of the formats listed for it.

Source selection	Clipboard contents, in order
Embedded object	Native OwnerLink Picture or other presentation format (optional) ObjectLink (included only if the server also supports links)
Linked object	OwnerLink Picture or other presentation format (optional; for linked objects, this can be Native data) ObjectLink
Pictorial data	Application-specific formats Native OwnerLink Picture ObjectLink
Structured data	Structured data formats (if selection is structured data only) Native OwnerLink Picture, text, and so on ObjectLink

Before copying data for an embedded or linked object to the clipboard, a server puts descriptions of the data formats on the clipboard. These data formats are listed in order of their level of description, from most descriptive to least. (For example, Microsoft Word would put rich-text format (RTF) onto the clipboard first, then the CF_TEXT clipboard format.)

When a user chooses the Paste command, the client application queries the formats on the clipboard and uses the first format that is compatible with the destination for the object. Because server applications put data onto the clipboard in order of their fidelity of description, the first acceptable format found by a client application is the best format for it to use. If the client application finds an acceptable format prior to the Native format, it incorporates the data into the target document without making it an embedded object. (For example, a Microsoft Word document would not make an embedded object from clipboard data that was in RTF format. Similarly, structured data or a structured document would be embedded into a drawing application but would be converted into the destination document's native data type if the destination were a worksheet or structured document.) If the client application cannot accept any of the data formats prior to Native and OwnerLink, it uses the Native and OwnerLink formats to make an embedded object and then finds an appropriate presentation format. The destination application may require different formats depending on where the selection is to be placed in the destination document; for example, pasting into a picture frame and pasting into a stream of text could require different formats.

When a user chooses the Paste Link command from the Edit menu, the client application looks for the ObjectLink format on the clipboard and ignores the Native and OwnerLink formats. The ObjectLink format identifies the source class, document, and object. If the application finds the ObjectLink format and a useful presentation format, it uses them to make an OLE link to the source document for the object. If the ObjectLink format is not available, the client application may look for the Link format and create a DDE link. This type of link does not support the OLE protocol.

When an application that does not support OLE copies from an OLE item on the clipboard, it ignores the Native, OwnerLink and ObjectLink formats; the behavior of the copying application does not change.

Registration

The registration database supports linked and embedded objects by providing a systemwide source of information about whether server applications support the OLE protocol, the names of the executable files for these applications, the verbs for classes of objects, and whether an object-handler library exists for a given class of object.

When a server application is installed, it registers itself as an OLE server with the registration database. (This database is supported by the dynamic-link library SHELL.DLL.) To register itself as an OLE server, a server application records in the database that it supports one or more OLE protocols. The only protocols supported by version 1.*x* of the Microsoft OLE libraries are StdFileEditing and StdExecute. StdFileEditing is the current protocol for linked and embedded objects. StdExecute is used only by applications that support the **OleExecute** function. (A third name, Static, describes a picture than cannot be edited by using standard OLE techniques.)

When a client activates a linked or embedded object, the client library finds the command line for the server in the database, appends the **/Embedding** or **/Embedding** *filename* command-line option, and uses the new command line to start the server. Starting the server with either of these options differs from the user starting it directly. Either a slash (/) or a hyphen (-) can precede the word Embedding. For details about how a server reacts when it is started with these options, see "Opening and closing objects."

The entries in the registration database are used whenever an application or library needs information about an OLE server. For example, client applications that support the Insert Object command refer to the database in order to list the OLE server applications that could provide a new object. The client application also uses the registration database to retrieve the name of the server application for the Paste Special dialog box.

Registration database

Applications typically add key and value pairs to the registration database by using Microsoft Windows Registration Editor (REGEDIT.EXE). Applications could also use the registration functions to add this information to the database.

The registration database stores keys and values as null-terminated strings. Keys are hierarchically structured, with the names of the components of the keys separated by backslash characters (\). The class name and server path should be registered for every class the server supports. (This class name must be the same string as the server uses when it calls the **OleRegisterServer** function.) If a class has an object-handler library, it should be registered using the **handler** keyword. An application should also register all the verbs its class or classes support. (An application's verbs must be sequential; for example, if an object supports three verbs, the primary verb is 0 and the other verbs must be 1 and 2.)

To be available for OLE transactions, a server should register the key and value pairs shown in the following example when it is installed. This example shows the form of key and value pairs as they would be added to a database with Registration Editor. Although the text string sometimes wraps to the next line in this example, the lines should not include newline characters when they are added to the database.

```
HKEY_CLASSES_ROOT\class name = readable version of class name
HKEY_CLASSES_ROOT\.ext = class name
HKEY_CLASSES_ROOT\class name\protocol\StdFileEditing\server =
    executable file name
HKEY_CLASSES_ROOT\class name\protocol\StdFileEditing\handler =
    dll name
HKEY_CLASSES_ROOT\class name\protocol\StdFileEditing\verb\0 =
    primary verb
HKEY_CLASSES_ROOT\class name\protocol\StdFileEditing\verb\1 =
    secondary verb
```

Servers that support the **OleExecute** function also add the following line to the database:

```
HKEY_CLASSES_ROOT\class name\protocol\StdExecute\server =
    executable file name
```

An ampersand (&) can be used in the verb specification to indicate that the following character is an accelerator key. For example, if a verb is specified as &Edit, the E key is an accelerator key.

A server can register the entire path for its executable file, rather than registering only the filename and arguments. Registering only the filename fails if the application is installed in a directory that is not mentioned in the PATH environment variable. Usually, registering the path and filename is less ambiguous than registering only the filename.

Servers can register data formats that they accept on calls to the **OleSetData** function or that they can return when a client calls the **OleRequestData** function. Clients can use this information to initialize newly created objects (for example, from data selected in the client) or when using the server as an engine (for example, when sending data to a chart and getting a new picture back). Client applications should not depend on the requested data format, because the calls can be rejected by the server.

In the following example, *format* is the string name of the format as passed to the **RegisterClipboardFormat** function or is one of the system-defined clipboard formats (for example, CF_METAFILEPICT):

```
HKEY_CLASSES_ROOT\class name\protocol\StdFileEditing
    \SetDataFormats = format[,format]
HKEY_CLASSES_ROOT\class name\protocol\StdFileEditing
    \RequestDataFormats = format[,format]
```

For compatibility with earlier applications, the system registration service also reads and writes registration information in the [embedding] section of the WIN.INI initialization file.

In the following example, the keyword **picture** indicates that the server can produce metafiles for use when rendering objects:

```
[embedding]
classname=comment,textual class name,path/arguments,picture
```

Version control for servers

Server applications should store version numbers in their Native data formats. New versions of servers that are intended to replace old versions should be capable of dealing with data in Native format that was created by older versions. It is sometimes important to give the user the option of saving the data in the old format, to support an environment with a mixture of new and old versions, or to permit data to be read by other applications that can interpret only the old format.

There can be only one application at a time (on one workstation) registered as a server for a given class name. The class name (which is stored with the Native data for objects) and the server application are associated in the registration database when the server application registers during installation.

If a new version of a server application allows the user to keep the old version available, a new class name should be allocated for the new server. A good way to do this is to append a version number to the class name. This allows the user to easily differentiate between the two versions when necessary. (The OLE libraries do not check these numbers.)

When the new version of the server is installed, the user should be given the option of either mapping the old objects to the new server (registering the new server as the server for both class names) or keeping them separate. When the user keeps them separate, the user will be aware of two kinds of object (for example, Graph1 and Graph2).

The user should be able to discard the old server version at a later time by remapping the registration database, typically with the help of the server setup program. To remap the database, the old and new objects are given the same value for *readable version of class name* (although their class names remain distinct). The OLE client library removes duplicate names when it produces the list in the Insert Object dialog box. When a client application produces a list by enumerating the registration database, the application must do this filtering itself.

Client user interface

When a user opens a document that contains a linked or embedded object, the client application uses the OLE functions to communicate with OLECLI.DLL. This library assists the client application with such tasks as loading and drawing objects, updating objects (when necessary), and interacting with server applications.

New and changed commands

An OLE client application typically implements the following new or changed commands as part of its Edit menu. (Although this user interface is not mandatory, it is recommended for consistency with existing OLE applications.)

Command	Description
Copy	Copies an object from a document to the clipboard.
Cut	Removes an object from a document and places it on the clipboard.
Paste	Copies an object from the clipboard to a document.
Paste Link	Inserts a link between a document and the file that contains an object.
Class Name Object	Makes it possible for the user to activate the verbs for a linked or embedded object. The actual text used instead of the *Class Name* placeholder depends upon the selected object.
Links	Makes it possible for the user to change link updating options, update linked objects, cancel links, repair broken links, and activate the verbs associated with linked objects.
Insert Object	Starts the server application chosen by the user from a dialog box and embeds in a document the object produced by the server. This command is optional.
Paste Special	Transfers an object from the clipboard to a document or inserts a link to the object, using the data format chosen by the user from a dialog box. This command is optional.

In addition to the listed menu changes, client applications must also implement changes to their Copy and Cut commands. When a linked or embedded object is selected in the client application, the application can use the **OleCopyToClipboard** function to implement the Cut and Copy commands.

When the user chooses the Paste command, a client application should insert the contents of the clipboard at the current position in a document. If the clipboard contains an object, choosing this command typically embeds the object in the document.

When the user chooses the Paste Link command, the client library typically inserts a linked object at the current position in a document. The object is displayed in the document, but the Native data that defines that object is stored elsewhere.

If a user copies a linked object to the clipboard, other documents can use this object to produce a link to the original data.

The *Class Name* Object command allows the user to choose one of an object's verbs. If the selection in the document is an embedded object, the *Class Name* placeholder is typically replaced by the class and name of the object; for example, if a user selects an object that is a range of spreadsheet cells for Microsoft Excel, the text of the command might be "Microsoft Excel Worksheet Object." If an object supports only one verb, the name of the verb should precede the class name in the menu item; for example, if the only verb for a text object is Edit, the text of the command might be "Edit WPDocument Object." When an object supports more than one verb, choosing the *Class Name* Object command brings up a cascading menu listing each of the verbs.

For more information about verbs, see "Verbs."

Choosing the Links command brings up a Links dialog box, which lists the selected links and their source documents and gives the user the opportunity to change how the links are updated, cancel the link, change the link, or activate the verbs for the link. A user can use this dialog box to repair links to objects that have been moved or renamed.

When the user chooses the Paste Special command, a client application should bring up a dialog box listing the data formats the client supports that are presently on the clipboard. The Paste Special dialog box makes if possible for the user to override the default behaviors of the Paste and Paste Link commands. For example, if the first format on the clipboard can be edited by the client application, the default behavior is for the client to copy the data into the document without making it into an object. The user could override this default behavior and create an object from such data by using the Paste Special command.

When the user chooses the Insert Object command, a client application should allow the user to insert an object of a specified class at the current position in a document. For example, to insert a range of spreadsheet cells in a text document, a user could choose the Insert Object command and select "Microsoft Excel Worksheet" from the dialog box. Selecting this item would start Microsoft Excel. The user would use Microsoft Excel to create the object to be embedded in the text document. When finished, the user would quit Microsoft Excel; the range of spreadsheet cells would automatically be embedded in the text document.

The Insert Object command is optional because a user could achieve the same results without it, although the procedure is less convenient. To use the same example as that shown in the preceding paragraph, the user could leave the client application, start Microsoft Excel, and use the Microsoft Excel Cut or Copy command to transfer data to the clipboard. After returning to the client application, the user could choose the Paste command to move the data from the clipboard into the text document.

If the user chooses the Undo command after activating an object, all the changes made since the object was last updated (or since the object was activated, if it has not been updated) are discarded and the object returns to its state prior to the update. The Undo command closes the connection to the server.

Using packages

A package is an embedded graphical object that contains another object, which can be linked or embedded. For example, a user can package a file in an icon and embed the icon in an OLE document. Most of the packaging capabilities are provided by the dynamic-link library SHELL.DLL.

A user can put a package into an OLE document in a number of different ways:

- Copy a file from File Manager to the clipboard, and then choose the Paste or Paste Link command from the Edit menu in the client application.
- Drag a file from File Manager and drop it in the open window for a document in a client application.
- Select Package from the list of objects in the Insert Object dialog box. This starts Object Packager, with which the user can associate a file or data selection with an icon or graphic. Choosing Update and then Exit from Object Packager's File menu puts the package in the client document.
- Run Packager directly, following the steps outlined in the previous list item.

A user whose system does not include the Windows version 3.1 File Manager can follow these steps to create a package by using Object Packager:

- Copy to the clipboard the data to be packaged.
- Open Object Packager and paste the data into it. (At this point, the user could modify the default icon, the default label identifying the icon, or both.)

- Choose Copy Package from the Object Packager Edit menu to copy the package to the clipboard.
- Choose the Paste command from the Edit menu in the client application to embed the package.

Server user interface

A server for linked and embedded objects is any application that can be used to edit an object when the OLE libraries inform it that the user of a client application has activated the object. (Some servers can use verbs other than Edit to work with an object.) Although client applications implement many changes to the user interface to support OLE, the user interface does not change significantly for server applications.

OLE servers typically implement changes to the following commands in the Edit menu. (Although this user interface is not mandatory, it is recommended for consistency with existing OLE applications.)

Command	Description
Cut	Transfers data from the application to the clipboard, deleting the data from the source document. A client application can use this data to create an embedded object.
Copy	Transfers a copy of the data from the application to the clipboard. A client application can use this data to create an embedded object and may be able to establish a link to the source document.

Some menu items change names or behave differently when a server is started as part of activating an object from within a compound document. The exact behavior of the server depends on whether the server supports the multiple document interface (MDI).

Updating objects from multiple-instance servers

When an embedded object is edited or played by a multiple-instance server—that is, a server that does not support the multiple document interface (MDI), the Save command on the File menu should change to Update. (This change does not occur when a server starts for a linked object.) When the user chooses the Update command, the object in the client is updated but the focus remains with the server window. To close the server window, the user chooses the Exit command.

When the user chooses the Save As, New, or Open command, the application should display a warning message asking the user whether to update the object in the compound document before performing the action. The New and Open commands break the link between the client and server applications. The Save As command also breaks the link between the client and server if the server was editing an embedded object.

Updating objects from single-instance servers

The same rules for updating objects from multiple-instance servers apply to single-instance (MDI) servers, with the following differences:

- When the focus in an MDI server changes from a window in which an embedded object was activated to a window in which a document that does not contain an embedded object is being edited, the Update command should change back to Save.
- When the user chooses the New or Open command, the window containing the embedded object remains open. (This eliminates the need to prompt the user to update the object.)

Object storage formats

The presentation data in linked or embedded objects can be thought of as a presentation object. A presentation objects can be standard, generic, or NULL. A standard presentation object is used when the format is metafile, bitmap, or device-independent bitmap (DIB). The client library supports the presentation objects, including drawing them. Neither client applications nor object handlers can use the presentation objects; they are solely for the use of the client library.

The following list gives the storage format for strings in OLE. The items appear in the order listed.

Type	Description
LONG	Length of string, including terminating null character.
Variable	Null-terminated stream of bytes.

The following list gives the storage format for the standard presentation object used for linked and embedded objects. The items appear in the order listed.

Type	Description
LONG	OLE version number.
LONG	Format identifier. This value is 5.
Variable	Class string. For standard presentation objects, this string is METAFILEPICT, BITMAP, or DIB.
LONG	Width of object, in MM_HIMETRIC units.
LONG	Height of object, in MM_HIMETRIC units.
LONG	Size of presentation data, in bytes.
Variable	Presentation data.

The following list gives the storage format for the generic presentation object used for linked and embedded objects. Generic objects are used when the clipboard format is other than metafile, bitmap, or DIB. The items appear in the order listed.

Type	Description
LONG	OLE version number.
LONG	Format identifier. This value is 5.
Variable	Class string.
LONG	Clipboard format value. If this value exists, the next item in storage is the size of the presentation data.
LONG	Clipboard format name. This value exists only if the clipboard format value is NULL.
LONG	Size of presentation data, in bytes.
Variable	Presentation data.

The following list gives the storage format for embedded objects. The items appear in the order listed.

Type	Description
LONG	OLE version number.
LONG	Format identifier. This value is 2.
Variable	Class string.
Variable	Topic string.
Variable	Item string.
LONG	Size of Native data, in bytes.
Variable	Native data.
Variable	Presentation object (standard, generic, or NULL).

The following list gives the storage format for linked objects. The items appear in the order listed.

Type	Description
LONG	OLE version number.
LONG	Format identifier. This value is 1.
Variable	Class string.
Variable	Topic string.
Variable	Item string.
Variable	Network name string.
short	Network type
short	Network driver version number.
LONG	Link update options.
Variable	Presentation object (standard, generic, or NULL).

The following list gives the storage format for static objects. The only difference between the format for static objects and the format for standard presentation objects is the value of the format identifier. The items appear in the order listed.

Type	Description
LONG	OLE version number.
LONG	Format identifier. This value is 3.
Variable	Class string. For static objects, this string is METAFILEPICT, BITMAP, or DIB.
LONG	Width of object, in MM_HIMETRIC units.
LONG	Height of object, in MM_HIMETRIC units.
LONG	Size of presentation data, in bytes.
Variable	Presentation data.

Client applications

A client application uses a server application to activate and render an object contained by a compound document. A client application provides storage for embedded objects, such contextual information as the target printer and page position, and a means for the user to activate the object and the server application associated with that object. Client applications also provide ways of putting embedded and linked objects into a document and taking them out again.

Client applications must provide permanent storage for objects in the compound document's file. When an item being saved is an embedded object, the client library stores the object's Native data, the presentation data for the object (for example, a metafile), and the OwnerLink information. When the item being saved is a link to another document, the client library stores the presentation data and the ObjectLink format.

Client applications accommodate asynchronous operations by defining a callback function to which the library sends notifications about current operations. As long as the client continues to dispatch messages, it can react to the notifications being sent to the callback function and to input from the user. For more information about asynchronous operations, see "Asynchronous operations."

Starting a client application

When a client application starts, it should follow these steps:

1. Register the clipboard formats that it requires.
2. Allocate and initialize as many **OLECLIENT** structures as required.
3. Allocate and initialize an **OLESTREAM** structure.

A client application can register the clipboard formats by calling the **RegisterClipboardFormat** function for each format, specifying such formats as Native, OwnerLink, ObjectLink, and any other formats it requires.

A client application uses two structures to receive information from the client library: **OLECLIENT** and **OLESTREAM**.

The **OLECLIENT** structure points to an **OLECLIENTVTBL** structure, which in turn points to a callback function supplied by the client application. The OLE libraries use this callback function to notify the client of any changes to an object. The parameters for the callback function are a pointer to the client structure, a pointer to the relevant object, and a value giving the reason for the notification. Typically, an application creates one **OLECLIENT** structure for each **OLEOBJECT** structure. Having a separate **OLECLIENT** structure for each object allows an application to take object-specific action in response to the OLE_QUERY_PAINT callback notification.

The **OLECLIENT** structure can also point to data that describes the state of an object. This data, when present, is supplied and used only by the client application. The client application allocates a separate **OLECLIENT** structure for each object and stores state information about that object in the structure. Because one argument to the callback function is a pointer to the **OLECLIENT** structure, this is an efficient method of retrieving the object's state information when the callback function is called.

The **OLESTREAM** structure points to an **OLESTREAMVTBL** structure, which is a table of pointers to client-supplied functions for stream input and output. The client libraries use these functions when loading and saving objects. A client can customize functions for particular situations, and a client can make such changes as varying the permanent storage for an object; for example, a client could store an object in a database, instead of in a file with the rest of the document.

The client application should create a pointer to the callback function in the **OLECLIENTVTBL** structure and pointers to the functions in the **OLESTREAMVTBL** structure by using the **MakeProcInstance** function. Callback functions should be exported in the module-definition file.

Opening a compound document

To open a compound document, a client application should take the following steps:

1. Register the document with the client library.
2. Load the document data from a file.
3. For each object in the document, call the **OleLoadFromStream** function.
4. List any objects with manual links so that the user can update them. Automatically update any automatic links.

The **OleRegisterClientDoc** function registers a document with the client library and returns a handle that is used in object-creation functions and document-management functions. (This registration does not involve the registration database.)

A client application should call the **OleLoadFromStream** function for each object in the document that will be shown on the screen or otherwise activated. (It is often not necessary to load every object in a document immediately when the document is opened.) Parameters for this function include a pointer to the **OLECLIENT** structure, which is used to locate the client's callback function (and which is sometimes used by the client to store private state information for the object), and a pointer to the **OLESTREAM** structure. The library calls the **Get** function in the **OLESTREAMVTBL** structure to load the object.

Document management

A client application should notify the library when it opens, closes, saves, or renames a document, or causes a document to revert to a previously saved state. A client application can use the following functions to accomplish these tasks:

Function	Description
OleRegisterClientDoc	Registers an opened document with the library.
OleRenameClientDoc	Informs the library that a document has been renamed.
OleRevertClientDoc	Informs the library that a document has reverted to a previously saved state.
OleRevokeClientDoc	Informs the library that a document should be closed or no longer exists.
OleSavedClientDoc	Informs the library that a document has been saved.

A client application should also maintain a persistent name for each object. This name should be unique within the scope of the client document and should be stored with the document. This name is specified when the object is created and should persist when the document is saved and reopened. When a client uses the **OleRename** function to change the name of an object, the new name must also be unique and must be stored with the document.

Saving a document

A client application should follow these steps to save a document:

1. Save the data for the document in the document's file.
2. For each object in the document, call the **OleSaveToStream** function.
3. When the library confirms that all objects have been saved, call the **OleSavedClientDoc** function.

A client application can call the **OleQuerySize** function to determine the size of the buffer required to store an object before calling **OleSaveToStream**.

Closing a document

A client application should follow these steps to close a document:

1. For each object in the document, call the **OleRelease** function.
2. Use either the **OleRevertClientDoc** or the **OleSavedClientDoc** function to register the current state of the document with the library.
3. When the library confirms that all objects have been closed, call the **OleRevokeClientDoc** function.

Asynchronous operations

When a client application calls a function that invokes a server application, actions taken by the client and server can be asynchronous. For example, the actions of updating a document and closing a server are asynchronous. Whenever an asynchronous operation begins, the client library returns OLE_WAIT_FOR_RELEASE. When a client application receives this notification, it must wait for the OLE_RELEASE notification before it quits. If the client cannot take further action until the asynchronous operation finishes, it should enter a message-dispatch loop and wait for OLE_RELEASE. Otherwise, it should allow the main message loop to continue dispatching messages so that processing can continue.

An application can run only one asynchronous operation at a time for an object; each asynchronous operation must end with the OLE_RELEASE notification before the next one begins. The client's callback function must receive OLE_RELEASE for all pending asynchronous operations before calling the **OleRevokeClientDoc** function.

Some of the object-creation functions return OLE_WAIT_FOR_RELEASE. The client application can continue to work with the document while waiting for OLE_RELEASE, but some functions (for example, **OleActivate**) cannot be called until the asynchronous operation has been completed.

If an application calls a function for an object before receiving OLE_RELEASE for that object, the function may return OLE_BUSY. The server also returns OLE_BUSY when processing a new request would interfere with the processing of a current request from a client application or user. When a function returns OLE_BUSY, the client application can display a message reporting the busy condition at this point or it can enter a loop to wait for the function to return OLE_OK. (The OLE_QUERY_RETRY notification is also sent to the client's callback function when the server is busy; when the callback function returns FALSE, the transaction with the server is ended.) Note that if the server uses the **OleBlockServer** function to postpone OLE activities, the OLE_QUERY_RETRY notification is not sent to the client.

The following example shows a message-dispatch loop that allows a client application to transact messages while waiting for the OLE_RELEASE notification:

```
while ((olestat = OleQueryReleaseStatus(lpObject)) == OLE_BUSY) {
   if (GetMessage(&msg, NULL, NULL, NULL)) {
       TranslateMessage(&msg);
       DispatchMessage(&msg);
   }
}
if (olestat == OLE_ERROR_OBJECT) {
   .
   .  /* The lpObject parameter is invalid. */
   .
}
else { /* if olestat == OLE_OK */
   .
   . /* The object is released, or the server has terminated. */
   .
}
```

A server application could end unexpectedly while a client is waiting for OLE_RELEASE. In this case, the client library recovers properly only if the client uses the **OleQueryReleaseStatus** function, as shown in the preceding example.

The following table shows which OLE functions can return the OLE_WAIT_FOR_RELEASE or OLE_BUSY value to a client application:

Function	OLE_BUSY	OLE_WAIT_FOR_RELEASE
OleActivate	Yes	Yes
OleClose	Yes	Yes
OleCopyFromLink	Yes	Yes
OleCreate	No	Yes
OleCreateFromClip	No	Yes
OleCreateFromFile	No	Yes
OleCreateFromTemplate	No	Yes
OleCreateLinkFromClip	No	Yes
OleCreateLinkFromFile	No	Yes
OleDelete	Yes	Yes
OleExecute	Yes	Yes
OleLoadFromStream	No	Yes
OleObjectConvert	Yes	No
OleReconnect	Yes	Yes
OleRelease	Yes	Yes
OleRequestData	Yes	Yes
OleSetBounds	Yes	Yes
OleSetColorScheme	Yes	Yes
OleSetData	Yes	Yes
OleSetHostNames	Yes	Yes
OleSetLinkUpdateOptions	Yes	Yes
OleSetTargetDevice	Yes	Yes
OleUnlockServer	No	Yes
OleUpdate	Yes	Yes

Displaying and printing objects

When an object has been loaded and, if necessary, brought up to date, the object can be displayed or printed with the container document. To display an object, the client application should set up the device context and bounding rectangle (ensuring that they use the same mapping mode) and then call the **OleDraw** function. The client application can use the **OleQueryBounds** function to retrieve the size of the bounding rectangle on the target device.

An object handler can be used to draw an object. If an object handler exists for an object, the call to the **OleDraw** function is received and processed by the object handler. If there is no object handler, the client library uses the object's presentation data to display or print the object.

If the presentation data for an object is a metafile, the library periodically sends an OLE_QUERY_PAINT notification to the client's callback function while drawing the object. If the callback function returns FALSE, the **OleDraw** function returns immediately and the drawing is ended. A client could also use the OLE_QUERY_PAINT notification to take some actions within the callback function and then return TRUE to indicate that drawing should continue. Any actions the client takes at this time should not interfere with the drawing operation; for example, the client should not scroll the window.

If the target device for an object changes (for example, when the user changes printers), the client application should call the **OleSetTargetDevice** function. The client should also call **OleSetTargetDevice** whenever an object is created or loaded.

If the size of the presentation rectangle for the object changes (for example, through action by the user) the client application should call the **OleSetBounds** function. After calling **OleSetBounds**, the client should call the **OleUpdate** function to update the object and then **OleDraw** to redisplay it.

Opening and closing objects

When the user requests the client application to activate an object, the client should check whether the object is busy by calling the **OleQueryReleaseStatus** function. If the object is busy, the client should either refuse the request to open the object or enter a message-dispatch loop, waiting for the OLE_RELEASE notification.

If the object to be activated is not busy, the client should call the **OleActivate** function. The library notifies the client when the server is open or when an error occurs.

The **OleActivate** function allows the client application to specify whether to display the activated object in a window of the server application. A client might hide the server window if an object is updated automatically.

A client application can use the **OleQueryOpen** function to determine whether a specified object is open. The **OleClose** function allows the client to close an open object. Closing an object terminates the connection with the server. To reestablish a terminated connection between a linked object and an open server, the client can use the **OleReconnect** function. To close an open object and release it from memory, a client application can call the **OleRelease** function.

The first time a client application activates a particular embedded object, the client should call the **OleSetHostNames** function, specifying the string the server window should display in its title bar. This string should be the name of the client document containing the object. The client does not need to call **OleSetHostNames** every time an embedded object is activated, because the library maintains a record of the specified names.

Deleting objects

To permanently delete an object from a document, the client should call the **OleDelete** function. **OleDelete** closes the specified object, if necessary, before deleting it.

Client Cut and Copy commands

A client application can copy an object to the clipboard by simply opening the clipboard, calling the **OleCopyToClipboard** function, and closing the clipboard again. If the client supports delayed rendering, however, it should follow these steps to cut or copy an object to the clipboard:

1. Open and empty the clipboard.

2. Put the preferred data formats on the clipboard.
3. Call the **OleEnumFormats** function to retrieve the formats for the object.
4. Call the **SetClipboardData** function to put the formats on the clipboard, specifying NULL for the handle of the data.

   If the call to the **OleEnumFormats** function retrieves the ObjectLink format, the client should call **SetClipboardData** with OwnerLink instead of ObjectLink format. (For more information, see the following description of the **OleCopyToClipboard** function.)
5. Put any additional presentation data formats on the clipboard.
6. Close the clipboard.

To support the Cut command on the Edit menu, an application can call **OleCopyToClipboard** and then delete the object by using the **OleDelete** function. (The client can put only one of the selected objects on the clipboard, even when the user has selected and cut or copied multiple objects. In this case, the client typically puts the first object in the selection onto the clipboard.)

The **OleCopyToClipboard** function always copies OwnerLink format, not ObjectLink format, to the clipboard. For embedded objects, Native data always precedes the OwnerLink format. If a linked object uses Native data, OwnerLink format always precedes the Native data. If an application uses the **OleGetData** function to retrieve data from a linked object that has been copied by using **OleCopyToClipboard**, it should specify ObjectLink format, not OwnerLink format, even if OwnerLink format was put on the clipboard.

When an application that can act as both a client and server copies a selection to the clipboard that contains one or more objects, it should first allocate enough memory for the selection. To discover how much memory is required for each object, the application can call the **OleQuerySize** function. When memory has been allocated, the application should call the **OleRegisterClientDoc** function, specifying Clipboard for the document name. (In this case, the handle returned by the call to **OleRegisterClientDoc** identifies a document that is used only during the copy operation.) To save each object to memory, the application calls the **OleClone** function, calls the **OleSaveToStream** function for the cloned object, and then calls the **OleRelease** function to free the memory for the cloned object. When the selection has been saved to the stream, the application can call the **SetClipboardData** function. If **SetClipboardData** is successful, the application should call the **OleSavedClientDoc** function. The application then calls the **OleRevokeClientDoc** function, specifying the handle retrieved by the call to **OleRegisterClientDoc.** For more information about the Cut and Copy commands, see "Server Cut and Copy commands."

Creating objects

A client application can put linked and embedded objects in a document by pasting them from the clipboard, creating them from a file, copying them from other objects, or by starting a server application to create them directly.

Object-creation functions

Each of the following functions creates an embedded or linked object in a specified document:

Function	Description
OleClone	Creates an exact copy of an object.
OleCopyFromLink	Creates an embedded object that is a copy of a linked object.
OleCreate	Creates an embedded object of a specified class.
OleCreateFromClip	Creates an object from the clipboard. This function typically creates an embedded object.
OleCreateFromFile	Creates an object by using the contents of a file. This function typically creates an embedded object.
OleCreateFromTemplate	Creates an embedded object by using another object as a template.
OleCreateInvisible	Creates an object without displaying the server application to the user.
OleCreateLinkFromClip	Creates an object by using information on the clipboard. This function typically creates a linked object.
OleCreateLinkFromFile	Creates an object by using the contents of a file. This function typically creates a linked object.
OleObjectConvert	Creates an object that supports a specified protocol by converting an existing object.

Each of these functions requires a parameter that points to an **OLEOBJECT** structure when the function returns. Server applications often create an **OLEOBJECT** structure whenever an object is created; **OLEOBJECT** points to functions that describe how the server interacts with the object. Before the client library gives the client application a pointer to this structure, the library includes with the structure some internal information corresponding to the OwnerLink or ObjectLink data. This internal information allows the client library to identify the correct server when an OLE function such as **OleActivate** passes it a pointer to an **OLEOBJECT** structure. For more information about the **OLEOBJECT** structure, see "Starting a server application."

Each new object must have a name that is unique to the client document. Although meaningful object names can be helpful, some applications assign unique object names simply by incrementing a counter for each new object. For more information about object names, see "Document management."

If a client application implements the Insert Object command, it should use the registration database to find out what OLE servers are available and then list those servers for the user. When the user selects one of the servers and chooses the OK button, the client can use the **OleCreate** function to create an object at the current position.

The **OleCopyFromLink, OleCreate,** and **OleCreateFromTemplate** functions always create an embedded object. The other object-creation functions can create either an embedded object or a linked object, depending on the order and type of available data.

If a client application's callback function receives the OLE_RELEASE notification after the client calls the **OleCreate** or **OleCreateFromFile** function, the client should respond by calling the **OleQueryReleaseError** function. If **OleQueryReleaseError** shows that there was an error when the object was created, the client application should delete the object.

Whenever an object-creation function returns OLE_WAIT_FOR_RELEASE, the calling application should either wait for the OLE_RELEASE notification or notify the user that the object cannot be created. For more information, see "Asynchronous operations."

If a client application accepts files dropped from File Manager, it should respond to the WM_DROPFILES message by calling the **OleCreateFromFile** function and specifying Packager for the lpszClass parameter.

Paste and Paste Link commands

A client application should follow these steps to create an embedded or linked object by pasting from the clipboard:

1. Call the **OleQueryCreateFromClip** function to determine whether to enable the Paste command. If this function fails when StdFileEditing is specified for the *lpszProtocol* parameter, call it again, specifying Static.
2. Call the **OleQueryLinkFromClip** function to determine whether to enable the Paste Link command.
   - If the user chooses the Paste command, open the clipboard and call the **OleCreateFromClip** function.
   - If the user chooses Paste Link, open the clipboard and call the **OleCreateLinkFromClip** function.
3. Close the clipboard.
4. Call the **OleQueryType** function to determine the kind of object created by the creation function. (Depending on the order of clipboard data, **OleCreateFromClip** can sometimes create a linked object and **OleCreateLinkFromClip** can sometimes create an embedded object.)

The client application should put the pasted data or object into the document at the current position. The client should select the object so that the user can work with it immediately. If both the **OleQueryCreateFromClip** and **OleQueryLinkFromClip** functions fail but there is data on the clipboard that the client can interpret, the client should enable the Paste command.

If the information on the clipboard is incomplete—for example, if Native data is not accompanied by the OwnerLink format—the Paste command should insert a static object into the document. (A static object consists of the presentation data for an object; it cannot be edited by using standard OLE techniques. Attempts to open static objects fail and generate no notifications.)

If the client application implements the Paste Special command, it should use the **EnumClipboardFormats** function to produce a list of data formats on the clipboard. The client should also check the registration database to find the full name of the server application. The Paste Link button in the Paste Special dialog box works in exactly the same way as the Paste Link command on the Edit menu.

If the DDE Link format is available on the clipboard instead of ObjectLink format, the client application should perform the same link operation that it supported prior to the implementation of OLE.

Undo command

A client application can use the **OleClone** function to support the Undo command. A cloned object is identical to the original except for connections to the server application; the cloned object is not automatically connected to the server. When the server is closed and the object is updated, the saved copy of the object gives the user the opportunity to undo all of the changes made in the server. Support for the Undo command is provided by the client application, because the server cannot maintain a record of the prior states of objects.

The Undo command restores an object to its condition prior to the last update from the server. To support this behavior, the client application must clone the object when it is first activated and then clone the updated object when an update occurs; the client must maintain two clones of the object. The clone of the original object must be maintained so that an updated object can be restored if the user chooses the Undo command. The clone of the updated object must be maintained to support the Undo command if the updated object is updated again. Because the data changes when the update occurs, the clone for supporting the Undo command must be made before any updates occur.

Because the client application cannot distinguish between different types of object activation, the client must clone an object for verbs that do not edit the object, even though no updates can occur in those cases.

Class Name Object command

A client application can implement the *Class Name* Object command by using the **OleActivate** function. **OleActivate** includes a parameter that allows the client to specify the verb chosen by the user.

Links command

When a user chooses the Links command, a dialog box appears listing every linked object in the document. The selected links are highlighted in the dialog box. The dialog box makes it possible for the user to invoke the verbs for an object, select whether link updating should be automatic or manual, update a link immediately, cancel a link, and repair broken links.

The Links dialog box includes buttons that allow the user to activate the primary and secondary verbs for an object. A client application can implement these buttons by using the **OleActivate** function.

A client application can use the **OleGetLinkUpdateOptions** and **OleSetLinkUpdateOptions** functions to support the link-update radio buttons in the Links dialog box. The following are the three possible update options:

Option	Description
oleupdate_always	Update the linked object whenever possible. This option supports the Automatic link-update radio button in the Links dialog box.
oleupdate_onsave	Update the linked object when the source document is saved by the server.
oleupdate_oncall	Update the linked object only on request from the client application. This option supports the Manual link-update radio button in the Links dialog box.

These update options control when updates to the presentation of an object occur. The contents of the source document are used to update the presentation whenever the link is activated.

To support the Update Now button in the Links dialog box, an application can call the **OleUpdate** function. When a user chooses Update Now, the client application should update the links the user selected.

A user's choosing the Cancel Link button in the Links dialog box changes an object into a picture that an application cannot edit by using standard OLE techniques. An application can implement the Cancel Link button by using the **OleObjectConvert** function.

A client application should activate the Change Link button in the Links dialog box only if all the selected links are to the same source document. When the client has the correct information, it can repair the link by using the **OleGetData** and **OleSetData** functions. To retrieve the link information for an object, a client can call the **OleGetData** function, specifying the ObjectLink format. (The call to **OleGetData** fails if ObjectLink is specified and the object is not a link.) A client can retrieve class information by using **OleGetData** and specifying either the OwnerLink format (for embedded objects) or the ObjectLink format (for linked objects). The client can make it possible for the user to edit the link information and store it in the object by using the **OleSetData** function, specifying the ObjectLink format.

Closing a client application

A client application should use the **OleRelease** function to remove all objects from memory when it shuts down. If the library returns the value OLE_WAIT_FOR_RELEASE instead of OLE_OK, the client should not quit. The client can perform many cleanup tasks while waiting for the OLE_RELEASE notification—for example, it can close files, free memory, and hide windows.

The OLE_RELEASE notification to the client's callback function indicates that an operation has finished in a server application, but it does not identify the operation or indicate whether the operation was successful. A client application can call the **OleQueryReleaseStatus** function to determine whether an operation has been completed for a specified object. The **OleQueryReleaseMethod** function indicates the nature of the operation that has finished for a specified object. To discover the error value for the operation, the client can call the **OleQueryReleaseError** function.

If a client owns the clipboard when it quits, it should make sure that the data on the clipboard is complete and in the correct order.

Server applications

An OLE server supplies functions that the server library calls when a user works with an object. The server library, OLESVR.DLL, uses DDE commands to communicate with the client library. When the client application calls one of the functions in the OLE API, the client library informs the server library and the server library routes the request to the appropriate function in the server-supplied list of function pointers.

In addition to the specialized functions that the server creates and which are called by the server library, there are ten OLE functions that allow a server to control the library's ability to gain access to the server and the documents and objects it controls:

Function	Description
OleBlockServer	Queues requests to the server until the server calls the **OleUnblockServer** function.
OleRegisterServer	Registers the specified server with the library. Information registered includes the class name and instance and whether the server supports single or multiple instances.
OleRegisterServerDoc	Registers a document with the server library.
OleRenameServerDoc	Renames the specified document.
OleRevertServerDoc	Restores a document to a previously saved state, without closing the document.
OleRevokeObject	Revokes access to the specified object.
OleRevokeServer	Revokes access to the specified server, closing any documents and ending communication with client applications.
OleRevokeServerDoc	Revokes access to the specified document.
OleSavedServerDoc	Informs the library that a document has been saved. Calling this function is equivalent to sending the OLE_SAVED notification.
OleUnblockServer	Processes a request from a queue created when the server application called the **OleBlockServer** function.

The **OleRevokeServer** and **OleRevokeServerDoc** functions can return OLE_WAIT_FOR_RELEASE. When a server application receives this error value, it should take the same action as a client application, dispatching messages until the server library calls the corresponding **Release** function.

Starting a server application

When a server application starts, it should follow these steps:

1. Register window classes and window procedures for the main window, documents, and objects.
2. Initialize the function tables for the **OLESERVERVTBL**, **OLESERVERDOCVTBL**, and **OLEOBJECTVTBL** structures.
3. Register the clipboard formats.
4. Allocate memory for the **OLESERVER** structure.
5. Register the server with the library by calling the **OleRegisterServer** function.
6. Check for the **/Embedding** and **/Embedding** *filename* options on the command line and act according to the following guidelines. (Applications should also check for **-Embedding** whenever they check for these options.)

- If neither **/Embedding** nor **/Embedding** *filename* is present, call the **OleRegisterServerDoc** function, specifying an untitled document.
- If the **/Embedding** option is present, do not register a document or display a window. (In this case, the server takes actions only in response to calls from the server library.)
- If the **/Embedding** *filename* option is present, do not display a window. Process the filename string and call the **OleRegisterServerDoc** function.

The **OLESERVERVTBL**, **OLESERVERDOCVTBL**, and **OLEOBJECTVTBL** structures are tables of function pointers. The server library uses these structures to route requests from the client application to the server. The server application should create the function pointers in these structures by using the **MakeProcInstance** function. The functions should also be exported in the application's module-definition file.

The **OLESERVER** structure contains a pointer to an **OLESERVERVTBL** structure. The **OLESERVERVTBL** structure contains pointers to functions that control such fundamental server tasks as opening files, creating objects, and terminating after an editing session. Several of the functions pointed to by the **OLESERVERVTBL** structure cause the server to allocate and initialize an **OLESERVERDOC** structure.

The **OLESERVERDOC** structure contains a pointer to an **OLESERVERDOCVTBL** structure. The **OLESERVERDOCVTBL** structure contains pointers to functions that control such tasks as saving or closing documents or setting document dimensions. The **OLESERVERDOCVTBL** structure also contains a function that causes the server to allocate and initialize an **OLEOBJECT** structure.

The **OLEOBJECT** structure contains a pointer to an **OLEOBJECTVTBL** structure. The **OLEOBJECTVTBL** structure contains pointers to functions that operate on objects. After the server application creates an **OLEOBJECT** structure, the server library gives information about the structure to the client library. The client library then creates a parallel **OLEOBJECT** structure (including internal information identifying the server application, the document, and the item for the object) and passes a pointer to that structure to the client application.

This hierarchy of structures—**OLESERVER**, **OLESERVERDOC**, and **OLEOBJECT**—makes it possible for a server to open as many documents as the library requests and for each document to contain as many objects as necessary.

A server application can register the clipboard formats by calling the **RegisterClipboardFormat** function for each format, specifying Native, OwnerLink, ObjectLink, and any other formats it requires.

When the server application starts, it creates an **OLESERVER** structure and then registers it with the library by calling the **OleRegisterServer** function. When this function returns, one of its parameters points to a server handle. The library uses this handle of refer to the server, and the server uses it in calls to the server-specific OLE functions.

If an OLE server application is also a DDE server, the class name specified in the call to the **OleRegisterServer** function cannot be the same as the name of the executable file for the application.

When a client working with a compound document opens a linked or embedded object for editing, the client library starts the server using the **/Embedding** command-line option. The server uses this option to determine whether the object has been opened directly by a user or as part of an editing session for linked and embedded objects. (If the object is a linked object, the **/Embedding** option is followed by a filename.) When a server is started for an embedded object with the **/Embedding** option, the server should not create a document or show a window. Instead, it should call the **OleRegisterServer** function and then enter a message-dispatch loop. (If the server is started with the **/Embedding** *filename* option, it should also call the **OleRegisterServerDoc** function.) The server then takes actions in response to calls from the library. The server should not make itself visible until the library calls the **Show** or **DoVerb** function in the **OLEOBJECTVTBL** structure. (Server applications should check for both **–Embedding** and **/Embedding**.)

By calling the **OleBlockServer** function, a server application can cause requests from the client library to be saved in a queue. When the server is ready for the server library to process the requests, it can call the **OleUnblockServer** function. It is best to use the **OleUnblockServer** function prior to the **GetMessage** function in a message loop, so that all blocked requests are unblocked before getting the next message. (Often a server returns OLE_BUSY instead of calling **OleBlockServer.** Returning OLE_BUSY has two advantages: It allows the client to decide whether to retry the message or discontinue the operation, and it allows the server to choose which requests to process.)

When an error occurs in a server-supplied function, the server should return the **OLESTATUS** error value that best describes the error. The OLE libraries use these error values to help determine the appropriate behavior in error situations. However, the client application does not necessarily receive the error values the server returns; the OLE libraries may change error values before passing them to the client application.

Opening a document or object

Whenever the server library calls the **Open**, **Create**, **CreateFromTemplate**, or **Edit** function in the **OLESERVERVTBL** structure, the server creates an **OLESERVERDOC** structure. If the document is opened by a call from the server library, the server application returns the **OLESERVERDOC** structure to the library. If the document is opened directly by a user, however, the server should call the **OleRegisterServerDoc** function to register the document with the library. The library then uses the **GetObject** function in the **OLESERVERDOCVTBL** structure to request the server to create an **OLEOBJECT** structure for each object requested by the client application.

A new instance of the server application is typically started when the client activates a linked or embedded object. This new instance is unnecessary if the object is already open in an instance of the server or if the server is a single-instance (MDI) server that is already open.

Whether the server library starts a new instance of a server to edit an embedded or linked object depends upon the value specified when the server calls the **OleRegisterServer** function.

Server Cut and Copy commands

A server application should follow these steps to cut or copy onto the clipboard data that a client can then use to create an embedded or linked object:

1. Open and empty the clipboard.
2. Put the data formats that describe the selection on the clipboard, using the **SetClipboardData** function.
3. Close the clipboard.

If the server cuts data onto the clipboard, rather than copying it, the server typically does not offer ObjectLink or Link formats, because the source for the data has been removed from the document.

The server should put data on the clipboard in the order given in "Clipboard conventions."

Typically, the server puts server-specific formats, Native format, OwnerLink format, and presentation formats on the clipboard. If it can support links, the server also puts ObjectLink format and, when appropriate, Link format on the clipboard. The server must provide a presentation format (CF_METAFILE, CF_BITMAP, or CF_DIB) if the server does not have an object handler. Native data can be used as a presentation format only if the server has an object handler that can use the Native data.

If a user copies onto the clipboard a selection that includes an embedded object or a link, the data formats the server should copy depend upon whether the container document modifies the object or link. If the document does not modify the object or link, the best formats are the Native and OwnerLink formats from the original source of the object. If the document modifies the object or link—for example, by recoloring it—the best formats are the Native and OwnerLink formats from the container document.

If a server uses a metafile as the presentation format for an object, the mapping mode for that metafile must be MM_ANISOTROPIC. When a server application uses fonts in these metafiles, it can improve performance by using TrueType fonts. (Metafiles scale better when they use TrueType fonts.) To use TrueType fonts exclusively, the server should set bit 2 (04h) of the **lpPitchAndFamily** member of the **LOGFONT** structure.

The OLE libraries express the size of every object in MM_HIMETRIC units. Neither the width nor height of an object should exceed 32,767 MM_HIMETRIC units.

Update, Save As, and New commands

When a server is started as part of editing an object from within a compound document, the server application should change the Save command on the File menu to Update. When the user chooses the Update command, the server should call the **OleSavedServerDoc** function.

When the user chooses the Save As, New, or Open command in a single-document server, the application should display a message asking the user whether to update the object in the compound document before performing the action. When the user chooses the Save As command, the server should call the **OleRename-ServerDoc** function. If the user responds to the message by choosing to save changes in the object before renaming the document, the server should call the **OleSavedServerDoc** function before calling **OleRenameServerDoc.** For embedded objects, choosing the Save As command causes the connection with the client to be broken, because this command reassociates a document in memory with the specified new file. For linked objects, calling **OleRenameServerDoc** when the user chooses Save As makes it possible for the client to associate the link with the new file.

Most server applications maintain a "dirty" flag that records whether changes have been made to each open document in an instance. The following table shows the rules that apply to this flag when the server edits an embedded object. By following these rules, a server can ensure that this flag is TRUE when the document being edited in the server matches the embedded object in the client and that, otherwise, this flag is FALSE.

Flag	Condition
TRUE	Library calls the **Create** function in the **OLESERVERVTBL** structure.
TRUE	Library calls the **CreateFromTemplate** function in **OLESERVERVTBL.**
TRUE	Document is changed in server.
FALSE	Library calls the **Edit** function in **OLESERVERVTBL.**
FALSE	Library calls the **GetData** function in **OLEOBJECTVTBL** with the Native data format. (The flag should not change for any other formats.)

A server following these rules displays the message asking whether to update the object whenever it destroys a document that was editing an embedded object and the "dirty" flag is TRUE.

In an MDI server application, the New and Open commands on the File menu simply open a new window, and the connection with the client application remains unchanged. The user can continue to work with the server application after choosing one of these commands, but when the user exits the server application, the focus does not necessarily return to the client application.

Typically, a server can call the **OleSavedServerDoc** function whenever an object needs to be updated in the client document, including when the server closes the document. When the server closes the document and the object should be updated, the server sends the OLE_CLOSED notification. Client applications receive the OLE_CLOSED notification for embedded objects but not for linked objects, because the server library intercepts the notification for linked objects.

Closing a server application

The server library calls the **Exit** function in the **OLESERVERVTBL** structure when the server must quit. The server library calls the **Release** function to inform the server that it is safe to quit; the server does not necessarily stop when the library calls **Release**.

The server must exit when it is invisible and the library calls **Release**. (The only exception is when an application supports multiple servers; in this case, an invisible server is sometimes not revocable when the library calls **Release**.) If the server has no open documents and it was started with the **/Embedding** option (indicating that it was started by a client application), the server should exit when the library calls the **Release** function. If the user explicitly loads a document into a single-instance (MDI) server, however, the server should not exit when the library calls **Release**.

When the user closes a server that has edited an embedded object without updating changes to the client application, the server should display a message asking whether to save the changes. If the user chooses to save the changes, the server should send the OLE_CLOSED notification and call the **OleRevokeServerDoc** function. (Because sending OLE_CLOSED prompts the server library to send data to the client library, it is not necessary to send OLE_CHANGED or OLE_SAVED. If the user chooses not to save the changes, the server should simply call the **OleRevoke-ServerDoc** function (without sending OLE_CLOSED).

A server can use the **OleRevokeObject** function to revoke a client's access to an object—for example, if the user destroys the object. Similarly, the **OleRevokeServerDoc** function revokes a client's access to a document. (Because **OleRevokeServerDoc** revokes a client's access to all objects in a document, an application that uses **OleRevokeServerDoc** does not need to call the **OleRevokeObject** function for objects in that document.) To terminate all conversations with client applications, the server can call the **OleRevokeServer** function. These functions inform the server library that the specified items are no longer available.

A server application can receive OLE_WAIT_FOR_RELEASE—for example, the **OleRevokeServerDoc** function can return this value. Although a server can enter a message-dispatch loop and wait for the library to call the server's **Release** function, servers should never enter message-dispatch loops inside any of the server-supplied functions that are called by the server library.

The client application should not instruct the server to close the document or exit when the server is editing a linked object, unless the server is updating the link without displaying the object to the user. Because a linked object exists independently of the client, the user controls saving and closing the document by using the server application.

If a server application owns the clipboard when it closes, it should make sure that the data on the clipboard is complete and in the correct order. For example, any Native data should be accompanied by the OwnerLink format.

Object handlers

An application developer can use object handlers to introduce customized features into implementations of linked and embedded objects. When an object handler exists for a class of object, the object handler supplants some or all of the functionality that is usually provided by the client library and the server application. The object handler can take specialized action for any of the functions it intercepts. The object handler passes functions that it does not take action on to the client library, which then implements the default processing for that class.

An application might use an object handler to render Native data as the presentation data for an object, instead of using metafiles or bitmaps. Object handlers could also be used to implement special behavior when an object is opened.

Implementing object handlers

A server installing an object handler registers the handler with the registration database, using the keyword **handler**. Whenever a client application calls one of the object-creation functions, the client library uses the class name specified for the object and the **handler** keyword to search the registration database. If the library finds an object handler, the client library loads the handler and calls it to create the object. The handler can create an object for which all of the creation functions and methods are defined by the handler, or it can call default object-creation functions in the client library.

The client library exports the object-creation OLE functions with new names; in each case, the prefix "Ole" is changed to "Def" (for "default"). Object handlers can import any of these functions and use them when creating objects.

Object handlers must import the following functions:

OLE function	Name exported by client library
OleCreate	DefCreate
OleCreateFromClip	DefCreateFromClip
OleCreateFromFile	DefCreateFromFile
OleCreateFromTemplate	DefCreateFromTemplate
OleCreateLinkFromClip	DefCreateLinkFromClip
OleCreateLinkFromFile	DefCreateLinkFromFile
OleLoadFromStream	DefLoadFromStream

When an object handler defines a function that is to be called by the client application, it should use the same name as the corresponding OLE function the client calls, with the prefix "Ole" replaced by "Dll". For example, when an object handler uses the **DefCreate** function exported by the client library, the handler should use it inside a function named **DllCreate**. When the client library finds an object handler for a class of object, it calls handler-specific object-creation functions by specifying this "Dll" prefix.

When the handler calls one of the default object-creation functions, it receives a handle of an **OLEOBJECT** structure, which in turn points to the **OLEOBJECTVTBL** structure containing the current object-management functions. The object handler should copy this **OLEOBJECTVTBL** structure and customize the structure by replacing any function pointers in the structure with pointers to functions of its own. (If the object handler saves the pointers to the default functions, any of the replacement functions can also call the default functions in the table of function pointers.) When the object handler has finished customizing the structure, it should replace the pointer to the old **OLEOBJECTVTBL** structure with a pointer to the modified **OLEOBJECTVTBL** structure.

When the client makes a call to a function in the client library, the call is dispatched through the object handler's **OLEOBJECTVTBL** structure. If the object handler has replaced the function pointer, the call is routed to the function supplied by the handler. Otherwise, the call is routed to the client library.

Each **OLECLIENT, OLEOBJECT, OLESERVER, OLESERVERDOC**, or **OLESTREAM** structure contains a pointer to a structure that contains a table of function pointers.

(Structures containing tables of function pointers are identified with the "VTBL" suffix.) Each of the structures containing a pointer to a "VTBL" structure can also contain extra instance-specific information. This information is meaningful only to the application that supplies it and should not be used by other applications; for example, an object handler should not attempt to use any instance-specific information in an **OLECLIENT** structure.

The object handler should use the "Def" and "Dll" renaming conventions when it defines specialized functions. For example, if an object handler modifies the **Draw** function from an object's **OLEOBJECTVTBL** structure, it should copy that **Draw** function to a function named **DefDraw** and replace the **Draw** function with a specialized function named **DllDraw.** Inside the **DllDraw** function, the object handler can call **DefDraw** if the default drawing operation is appropriate in a particular case.

The following example demonstrates this process of copying and replacing pointers to functions. Functions with the "Dll" prefix should be exported in the module-definition file.

```
/* Declare the DllDraw and DefDraw functions.              */

OLESTATUSFARPASCALDllDraw(LPOLEOBJECT,HDC,LPRECT,LPRECT,HDC);
OLESTATUS(FARPASCAL*DefDraw)(LPOLEOBJECT,HDC,LPRECT,LPRECT,HDC);

/* Copy the Draw function from OLEOBJECTVTBL to DefDraw. */

    DefDraw = lpobj->lpvtbl->Draw;

/* Copy DllDraw to OLEOBJECTVTBL.                          */

    *lpobj->lpvtbl->Draw = DllDraw;


OLESTATUSFARPASCALDllDraw(lpObject,hdc,lpBounds,lpWBounds,
     hdcFormat)
LPOLEOBJECT      lpObject;
HDC              hdc;
LPRECT           lpBounds;
LPRECT           lpWBounds;
HDC              hdcFormat;
{
    /* Return DefDraw if Native data is not available.   */

    if ((*lpobj->lpvtbl->GetData) (lpobj, cfNative, &hData) != OLE_OK)
        return (*DefDraw) (lpobj, hdc, lpBounds, lpWBounds, hdcFormat);
    .
    .
    .
}
```

Creating objects in an object handler

Most of the object-creation functions in the OLE API work in exactly the same way when they are renamed and used by object-handler DLLs. Two functions are somewhat different, however: **OleCreateFromClip** and **OleLoadFromStream.**

DefCreateFromClip and DllCreateFromClip

When the client library calls the **DllCreateFromClip** function, the library includes a parameter that is not specified in the original call to the **OleCreateFromClip** function. This parameter, *objtype,* specifies whether the object being created is an embedded object or a link; its value can be either OT_LINK or OT_EMBEDDED.

The following syntax block shows the *objtype* parameter when an object handler uses the **DefCreateFromClip** function. The **DllCreateFromClip** function has exactly the same syntax as **DefCreateFromClip**.

```
OLESTATUS DefCreateFromClip(lpszProtocol, lpclient, lhclientdoc,
    lpszObjname, lplpobject, renderopt, cfFormat, objtype);
LPSTR lpszProtocol;            /* address of string for protocol name  */
LPOLECLIENT lpclient;          /* address of client structure          */
LHCLIENTDOC lhclientdoc;       /* long handle of client document       */
LPSTR lpszObjname;             /* string for object name               */
LPOLEOBJECT FAR * lplpobject;  /* address of pointer to object         */
OLEOPT_RENDER renderopt;       /* rendering options                    */
OLECLIPFORMAT cfFormat;        /* clipboard format                     */
LONG objtype;                  /* OT_LINKED or OT_EMBEDDED             */
```

If **DllCreateFromClip** calls **DefCreateFromClip, DllCreateFromClip** should pass it the *objtype* parameter along with the other parameters from the version of **DefCreateFromClip** that was exported by the client library. **DllCreateFromClip** can modify some of these parameters before passing them back to **DefCreateFromClip**. For example, the object handler could specify a different value for the *renderopt* parameter when it calls **DefCreateFromClip**. If the client calls this function with **olerender_draw** for *renderopt* and the handler performs the drawing with Native data, the handler could change **olerender_draw** to **olerender_none**. If the client calls this function with **olerender_draw** for *renderopt* and the handler calls the **GetData** function and performs the drawing based on a class-specific format, the handler could change **olerender_draw** to **olerender_format**. If the handler needed a different rendering format than the format specified by the client application, the object handler could also change the value of the *cfFormat* parameter in the call to **DefCreateFromClip**.

If an object handler uses Native data to render an embedded object, the handler can call the library and specify **olerender_none.** If a handler uses Native data to render a linked object, it can use **olerender_format** and specify Native data. When the handler's **Draw** function is called, the handler calls the **GetData** function, specifying Native data, to do the rendering. If a handler uses a private data format, the procedure is the same—except that the private format is specified with the **olerender_format** option and with the **GetData** function.

DefLoadFromStream and DllLoadFromStream

When the client library calls the **DllLoadFromStream** function, the library includes three parameters that are not specified in the original call to the **OleLoadFromStream** function. One of the additional parameters is *objtype*, as described for **DefCreateFromClip** and **DllCreateFromClip**. The other two parameters are *aClass*, which is an atom containing the class name for the object, and *cfFormat*, which specifies a private clipboard format that the object handler can use for rendering the object.

The following syntax block shows the *objtype*, *aClass*, and *cfFormat* parameters when an object handler uses the **DefLoadFromStream** function. The **DllLoadFromStream** function has exactly the same syntax as **DefLoadFromStream**.

```
OLESTATUS DefLoadFromStream(lpstream, lpszProtocol, lpclient,
    lhclientdoc, lpszObjname, lplpobject, objtype, aClass, cfFormat);
LPOLESTREAM lpstream;           /* address of stream for object        */
LPSTR lpszProtocol;             /* address of string for protocol name  */
LPOLECLIENT lpclient;           /* address of client structure         */
LHCLIENTDOC lhclientdoc;        /* long handle of client document      */
LPSTR lpszObjname;              /* string for object name              */
LPOLEOBJECT FAR * lplpobject;   /* address of pointer to object        */
LONG objtype;                   /* OT_LINKED or OT_EMBEDDED            */
ATOM aClass;                    /* atom containing object's class name  */
OLECLIPFORMAT cfFormat;         /* private data format for rendering    */
```

If **DllLoadFromStream** calls **DefLoadFromStream**, **DllLoadFromStream** should pass it the three additional parameters along with the other parameters from the version of **DefLoadFromStream** that was exported by the client library.

DllLoadFromStream can modify some of these parameters before passing them back to **DefLoadFromStream**. For example, the object handler could modify the value of the *cfFormat* parameter to specify a private data format it would use to render the object.

When the client calls the object handler with **DefLoadFromStream**, the handler uses the **Get** function from the **OLESTREAMVTBL** structure to obtain the data for the object.

Direct use of Dynamic Data Exchange

The OLE libraries, OLECLI.DLL and OLESVR.DLL, use DDE messages to communicate with each other. Although client and server applications can use DDE directly, without employing OLECLI.DLL or OLESVR.DLL, this method of implementing OLE is not recommended. Future enhancements to the OLE libraries will benefit applications that use the libraries but will not benefit applications that use DDE directly.

The following information about the DDE-based OLE protocol is provided for applications that must implement DDE directly, despite losing the ability to take advantage of future enhancements to the system.

Implementation of the OLE protocol requires implementation of the underlying DDE protocol. All the standard DDE rules and facilities apply. Applications that conform to this protocol must also conform to the DDE specification. Conforming to this specification implies supporting the System topic and the standard items in that topic.

Client applications and direct use of Dynamic Data Exchange

When opening a link or an embedded document, the client application should look up the class name in the registration database, as described in "Registration."

The following pseudocode illustrates the chain of events for a client implementing OLE through DDE. Whenever a client that attempts to establish a conversation with a server receives responses from more than one server, the client should accept the first server and reject the others.

Linked object:

```
WM_DDE_INITIATE class name, document name
if not found {
    WM_DDE_INITIATE class name, OLESystem
    if not found {
        WM_DDE_INITIATE class name, System
        if not found {
            launch application name, /Embedding
            fLaunched = true
            WM_DDE_INITIATE class name, OLESystem
            if not found {
                WM_DDE_INITIATE class name, System
                if not found
                    return error
            }
        }
    }

    /*
     * Now there is a conversation with the server on the
     * System or OLESystem topic.
     */

    WM_DDE_EXECUTE StdOpenDocument(DocumentName)
    WM_DDE_INITIATE class name, document name
    if not found {
        if(fLaunched) WM_DDE_EXECUTE StdExit /* clean up */
            return error
    }
}

/*
 * Now there is a conversation with the correct document.
 */
```

Embedded object:

```
WM_DDE_INITIATE class name, OLESystem
if not found {
        WM_DDE_INITIATE class name, System
        if not found {
                launch application name, /Embedding
                fLaunched = true
                WM_DDE_INITIATE class name, OLESystem
                if not found {
                        WM_DDE_INITIATE class name, System
                        if not found
                                return error
                }
        }
}

/*
 * Now there is a conversation with the server on the system or
 * OLESystem topic.
 */

DDE_EXECUTE StdEditDocument(DocumentName)

/*
 * Or StdCreateDoc if this is an Insert Object command
 */

WM_DDE_INITIATE class name, document name
if not found {
        if(fLaunched) DDE_EXECUTE StdExit          /* clean up */
                return error
}

/* Now there is a conversation with the correct document. */
```

Server applications and direct use of Dynamic Data Exchange

When a server receives the **/Embedding** command-line argument, it should not create a new default document. Instead, it should wait until the client sends either the **StdOpenDocument** command or the **StdEditDocument** command followed by the Native data and then instructs the server to show the window. The server can use the **StdHostNames** item to display the client's name in the window title.

The following pseudocode illustrates the chain of events for a server implementing OLE through DDE. The example shows two cases: one in which the server reuses a single instance for editing all objects (in MDI child windows), and another in which a new instance is used for each object. Applications that use a new instance for each object should reject requests to open or create a new document when they already have a document open.

MDI application:

```
case WM_DDE_INITIATE:
    if class name == this class {
        if (DocumentName == OLESystem || DocumentName ==
System)
            WM_DDE_ACK
        else if DocumentName == name of some open document
            WM_DDE_ACK
    }
```

Multiple-instance application:

```
case WM_DDE_INITIATE:
    if class name == this class {
        if (DocumentName == OLESystem || DocumentName ==
System) {
            if no documents are open
                WM_DDE_ACK
        }
        else if DocumentName == name of some open document
            WM_DDE_ACK
    }
```

Conversations

Document operations are performed during conversations with an application's OLESystem or System topic. The document's class name is used to establish the conversation.

Data transfer and negotiation operations are performed during conversations with the document (that is, the topic). The document name is used to establish the conversation.

Note that the topic name is used only in initiating conversations and is not fixed throughout the conversation; permitting the document to be renamed does not mean that there will be two names. Therefore, it is reasonable to tie the topic name to the document name.

Items for the system topic

An application using DDE-based OLE can use three new items for the System topic: the Topics item, the Protocols item, and the Status item.

The Topics item returns a list of DDE topic names that the server application has open. Where topics correspond to documents, the topic name is the document name.

The Protocols item returns a list of protocol names supported by the application. The list is returned in tab-separated text format. A protocol is a defined set of DDE execute strings and item and format conventions that the application understands. The protocol currently defined for linked and embedded objects is the following:

Protocol: StdFileEditing *commands/items/formats*

For compatibility with client applications that were written before the implementation of the OLE protocol, server applications that use the DDE protocol directly should also include the string Embedding in the list of protocols.

The Status item is a text item that returns Ready if the server is prepared to respond to DDE requests; otherwise, it returns Busy. This item can be queried to determine if the client should offer such functions as one that gives the user an opportunity to update the object. Because it is possible that a server could reject or defer a request even if Status returns Ready, client applications should not depend solely on the Ready item.

Standard item names and notification control

Applications supporting OLE with direct DDE use four clipboard formats in addition to the regular data and picture formats. These are ObjectLink, OwnerLink, Native, and Binary. Binary format is a stream of bytes whose interpretation is implicit in the item; for example, the **EditEnvItems**, **StdTargetDevice**, and **StdHostNames** items are in Binary format. The ObjectLink, OwnerLink, and Native formats are described in "Clipboard conventions."

New items available on each topic other than the System topic are defined for this protocol. These items are the following:

Item	Description
StdDocumentName	Contains the permanent document name associated with the topic. If no permanent storage is associated with the topic, this item is empty. This item supports both request and advise transactions and can be used to detect the renaming of open documents.
EditEnvItems	Returns a list in tab-separated text format of the items that contain environmental information supported by the server for its documents. Currently defined items are **StdHostNames**, **StdDocDimensions**, and **StdTargetDevice.** Applications can declare other items (and define their interpretations if Binary format is used) to permit clients that are informed of these items to provide more detailed information. Servers that cannot use particular items should omit their names from the EditEnvItems item. Clients should use the WM_DDE_REQUEST message with this item to find out which items the server can use and should supply the data through a WM_DDE_POKE message.

StdHostNames	Accepts information about the client application, in Binary format interpreted as the following structure:
	`struct {`
	`    WORD clientNameOffset;`
	`    WORD documentNameOffset;`
	`    BYTE data[];`
	`}StdHostNames;`

Item	Description
	The offsets are relative to the start of the data array. They indicate the starting point for the appropriate information in the array.

StdTargetDevice	Accepts information about the target device that the client is using. This information is in Binary format, interpreted as the following structure. Offsets are relative to the start of the data array.
	`typedefstruct_OLETARGETDEVICE{`
	`WORD otdDeviceNameOffset;`
	`WORD otdDriverNameOffset;`
	`WORD otdPortNameOffset;`
	`WORD otdExtDevmodeOffset;`
	`WORD otdExtDevmodeSize;`
	`WORD otdEnvironmentOffset;`
	`WORD otdEnvironmentSize;`
	`BYTE otdData[];`
	`}OLETARGETDEVICE;`

StdDocDimensions	Accepts information about the size of a document. This information is in Binary format, interpreted as the following structure. These values are specified in MM_HIMETRIC units.
	`struct {`
	`int iXContainer;`
	`int iYContainer;`
	`}StdDocDimensions;`

StdColorScheme	Returns the colors that the server is currently using and accepts information about the colors that the client requests the server to use. This information is in Binary format, interpreted as a **LOGPALETTE** structure.
null	Specifies a request or advise transaction on all data contained in the topic. This item is a zero-length item name.

The update method used for advise transactions on items follows a convention in which an update specifier is appended to the actual item name. The item is encoded as follows:

itemname/update type

For backward compatibility, omitting the update type has the same result as specifying **/Change**. The *update type* placeholder may be filled with one of the following values:

Value	Meaning
/Change	Notify for each change.
/Close	Notify when document is closed.
/Save	Notify when document is saved.

DDE server applications are required to save each occurrence of a WM_DDE_ADVISE message that specifies a unique combination of *itemname, update type, format,* and *conversation.* A notification is disabled by a WM_DDE_UNADVISE message with corresponding parameters. If the WM_DDE_UNADVISE message does not specify a format, it disables the oldest notification in first in, first out (FIFO) rotation.

Standard commands in DDE execute strings

The syntax for standard commands sent in execute strings is the same as for other DDE commands:

command(argument1,argument2,...)[command2(argument1,argument2,...)]

Commands without arguments do not require parentheses. String arguments must be enclosed in double quotes.

International execute commands

DDE execute strings are typically sent from a macro language in an external application and are typically localized. OLE execute commands, however, are sent by application programs for their own purposes, need not be localized, and must be commonly recognized.

The OLE standard execute commands should not be localized; the U.S. spelling and separator characters are used. Therefore, the following rules apply:

- Client applications and the client library send standard execute commands in U.S. form.
- The server library must receive the U.S. form for these commands.
- Servers written directly to the DDE-level protocol should parse the U.S. form, if they have no additional commands.

- Servers that support both OLE and localized DDE execute commands should first parse the string by using localized separators. If this fails, they should parse it again using the U.S. form and, if successful, should execute the command. Optionally, if the command is received in the U.S. form, the server can check that the command is one of the valid standard commands.

Required commands

This section lists commands that must be supported by server applications.

The **StdNewDocument**, **StdNewFromTemplate**, **StdEditDocument**, and **StdOpenDocument** commands all make the document available for DDE conversations with the name *DocumentName*. They do not show any window associated with the document; the client must send the **StdShowItem** and **StdDoVerbItem** commands, or the **StdDoVerbItem** command alone to make the window visible. This enables the client to negotiate additional parameters with the server (for example, the **StdTargetDevice** item) without causing unnecessary repaints.

StdNewDocument(*ClassName, DocumentName***)**
Creates a new, empty document of the given class, with the given name, but does not save it. The server should return an error value if the document name is already in use. When the client receives this error, it should generate another name and try again.

The server should not show the window until it receives a **StdShowItem** command. Waiting for the client to send the **StdShowItem** and **StdDoVerbItem** commands makes it possible for the client to negotiate additional parameters (for example, by using **StdTargetDevice**) without forcing the window to repaint.

StdNewFromTemplate(*ClassName, DocumentName, TemplateName***)**
Creates a new document of the given class with the given document name, using the template with the given permanent name (that is, filename).

The server should not show the window until it receives a **StdShowItem** command. Waiting for the client to send a **StdShowItem** command makes it possible for the client to negotiate additional parameters (for example, by using **StdTargetDevice**) without forcing the window to repaint.

StdEditDocument(*DocumentName*)

Creates a document with the given name and prepares to accept data that is poked into it with WM_DDE_POKE. The server should return an error if the document name is already in use. When the client receives this error, it should generate another name and try again.

The server should not show the window until it receives a **StdShowItem** command. Waiting for the client to send a **StdShowItem** command makes it possible for the client to negotiate additional parameters (for example, by using **StdTargetDevice**) without forcing the window to repaint.

StdOpenDocument(*DocumentName*)

Sent to the System topic. This command opens an existing document with the given name.

The server should not show the window until it receives a **StdShowItem** command. Waiting for the client to send a **StdShowItem** command makes it possible for the client to negotiate additional parameters (for example, by using **StdTargetDevice**) without forcing the window to repaint.

StdCloseDocument(*DocumentName*)

Sent to the System topic. This command closes the window associated with the document. Following acknowledgment, the server terminates any conversations associated with the document. The server should not activate the window while closing it.

StdShowItem(*DocumentName, ItemName [, fDoNotTakeFocus]*)

Sent to the System topic. This command makes the window containing the named document visible and scrolls to show the named item (if any). The optional third argument indicates whether the server should take the focus and bring itself to the front. This argument should be TRUE if the server should not take the focus; otherwise, it should be FALSE. The default value is FALSE.

StdExit

Shuts down the server application. This command should be used only by the client application that launched the server. This command is available in the System topic only.

StdExit is sent to shut down an application if an error occurs during the startup phase or if the client started the server for an invisible update. If servers have unsaved data opened by the user, they should ignore this command.

Variants on required commands

The following variants of the above commands may be sent to the document topic rather than the System topic. This allows a client that already has a conversation with the document to avoid opening an additional conversation with the system. The document name is omitted from these commands because it is implied by the conversation topic and because it may have been changed by the server. This kind of name change does not invalidate the conversation. The client should not be forced to keep track of the name change unnecessarily. However, the server must be able to use the conversation information to identify the document on which to operate.

StdCloseDocument
Sent to the document conversation. This command closes the document associated with the conversation without activating it. This command causes a WM_DDE_TERMINATE message to be posted by the server window following the acknowledgment.

StdDoVerbItem(*ItemName, iVerb, fShow, fDoNotTakeFocus*)
Sent to the document conversation. This command is similar to the **StdShowItem** command, except that it includes an integer indicating which of the registered operations to perform and a flag indicating whether to show the window. The server can ignore the *fShow* flag, if necessary.

StdShowItem(*ItemName [, fDoNotTakeFocus]*)
Sent to the document conversation. This command shows the document window, scrolling if necessary to bring the item into view. If the item name is NULL, scrolling does not occur. The optional second argument indicates whether the server should take the focus and bring itself to the front. This argument should be TRUE if the server should not take the focus; otherwise, it should be FALSE. The default value is FALSE.

C H A P T E R

4

Functions

AbortDoc 3.1

Syntax int AbortDoc(hdc)

function AbortDoc(DC: HDC): Integer;

The **AbortDoc** function terminates the current print job and erases everything drawn since the last call to the StartDoc function. This function replaces the ABORTDOC printer escape for Windows version 3.1.

Parameters *hdc* Identifies the device context for the print job.

Return Value The return value is greater than or equal to zero if the function is successful. Otherwise, it is less than zero.

Comments Applications should call the **AbortDoc** function to terminate a print job because of an error or if the user chooses to cancel the job. To end a successful print job, an application should use the **EndDoc** function.

If Print Manager was used to start the print job, calling the **AbortDoc** function erases the entire spool job—the printer receives nothing. If Print Manager was not used to start the print job, the data may have been sent to the printer before **AbortDoc** was called. In this case, the printer driver would have reset the printer (when possible) and closed the print job.

See Also **EndDoc, SetAbortProc, StartDoc**

AbortProc 3.1

Syntax BOOL CALLBACK AbortProc(hdc, error)

TAbortProc = function(DC: HDC; Error: Integer): Bool;

The **AbortProc** function is an application-defined callback function that is called when a print job is to be canceled during spooling.

Parameters

hdc	Identifies the device context.
error	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 this parameter is SP_OUTOFDISK, the application need not cancel the print job. If it does not cancel the job, it must yield to Print Manager by calling the **PeekMessage** or **GetMessage** function.

Return Value The callback function should return TRUE to continue the print job or FALSE to cancel the print job.

Comments An application installs this callback function by calling the **SetAbortProc** function. **AbortProc** is a placeholder for the application-defined function name. The actual name must be exported by including it in an **EXPORTS** statement in the application's module-definition file.

See Also **GetMessage, PeekMessage, SetAbortProc**

AllocDiskSpace 3.1

Syntax #include <stress.h>
int AllocDiskSpace(lLeft, uDrive)

function AllocDiskSpace(1Left: Longint; wDrive: Word): Integer;

The **AllocDiskSpace** function creates a file that is large enough to ensure that the specified amount of space or less is available on the specified disk partition. The file, called STRESS.EAT, is created in the root directory of the disk partition.

If STRESS.EAT already exists when **AllocDiskSpace** is called, the function deletes it and creates a new one.

Parameters

lLeft	Specifies the number of bytes to leave available on the disk.
uDrive	Specifies the disk partition on which to create the STRESS.EAT file. This parameter must be one of the following values:

Value	Meaning
EDS_WIN	Creates the file on the Windows partition.
EDS_CUR	Creates the file on the current partition.
EDS_TEMP	Creates the file on the partition that contains the TEMP directory.

Return Value The return value is greater than zero if the function is successful; it is zero if the function could not create a file; or it is –1 if at least one of the parameters is invalid.

Comments In two situations, the amount of free space left on the disk may be less than the number of bytes specified in the *lLeft* parameter: when the amount of free space on the disk is less than the number in *lLeft* when an application calls **AllocDiskSpace**, or when the value of *lLeft* is not an exact multiple of the disk cluster size.

The **UnAllocDiskSpace** function deletes the file created by **AllocDiskSpace**.

See Also **UnAllocDiskSpace**

AllocFileHandles 3.1

Syntax

```
#include <stress.h>
int AllocFileHandles(Left)
```

```
function AllocFileHandles(left: Integer): Integer;
```

The **AllocFileHandles** function allocates file handles until only the specified number of file handles is available to the current instance of the application. If this or a smaller number of handles is available when an application calls **AllocFileHandles**, the function returns immediately.

Before allocating new handles, this function frees any handles previously allocates by **AllocFileHandles**.

Parameters

Left	Specifies the number of file handles to leave available.

Return Value The return value is greater than zero if **AllocFileHandles** successfully allocates at least one file handle. The return value is zero if fewer than the specified number of file handles were available when the application called **AllocFileHandles**. The return value is –1 if the *Left* parameter is negative.

Comments **AllocFileHandles** will not allocate more than 256 file handles, regardless of the number available to the application.

The **UnAllocFileHandles** function frees all file handles previously allocated by **AllocFileHandles**.

See Also **UnAllocFileHandles**

AllocGDIMem 3.1

Syntax

```
#include <stress.h>
BOOL AllocGDIMem(uLeft)
```

```
function AllocGDIMem(wLeft: Word): Bool;
```

The **AllocGDIMem** function allocates memory in the graphics device interface (GDI) heap until only the specified number of bytes is available. Before making any new memory allocations, this function frees memory previously allocated by **AllocGDIMem**.

Parameters

uLeft	Specifies the amount of memory, in bytes, to leave available in the GDI heap.

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

Comments The **FreeAllGDIMem** function frees all memory allocated by **AllocGDIMem**.

See Also **FreeAllGDIMem**

AllocMem 3.1

Syntax #include <stress.h>
BOOL AllocMem(dwLeft)

function AllocMem(dwLeft: Longint): Bool;

The **AllocMem** function allocates global memory until only the specified number of bytes is available in the global heap. Before making any new memory allocations, this function frees memory previously allocated by **AllocMem**.

Parameters *dwLeft* Specifies the smallest size, in bytes, of memory allocations to make.

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

Comments The **FreeAllMem** function frees all memory allocated by **AllocMem**.

See Also **FreeAllMem**

AllocUserMem 3.1

Syntax #include <stress.h>
BOOL AllocUserMem(uContig)

function AllocUserMem(wContig: Word): Bool;

The **AllocUserMem** function allocates memory in the USER heap until only the specified number of bytes is available. Before making any new allocations, this function frees memory previously allocated by **AllocUserMem**.

Parameters *uContig* Specifies the smallest size, in bytes, of memory allocations to make.

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

Comments The **FreeAllUserMem** function frees all memory allocated by **AllocUserMem**.

See Also **FreeAllUserMem**

CallNextHookEx 3.1

Syntax LRESULT CallNextHookEx(hHook, nCode, wParam, lParam)

function CallNextHookEx(Hook: HHook; Code: Integer; wParam: Word; lParam: Longint): Longint;

The **CallNextHookEx** function passes the hook information to the next hook function in the hook chain.

Parameters

hHook	Identifies the current hook function.
nCode	Specifies the hook code to pass to the next hook function. A hook function uses this code to determine how to process the message sent to the hook.
wParam	Specifies 16 bits of additional message-dependent information.
lParam	Specifies 32 bits of additional message-dependent information.

Return Value The return value specifies the result of the message processing and depends on the value of the *nCode* parameter.

Comments Calling the **CallNextHookEx** function is optional. An application can call this function either before or after completing any processing in its own hook function. If an application does not call **CallNextHookEx**, Windows will not call the hook functions that were installed before the application's hook function was installed.

See Also **SetWindowsHookEx, UnhookWindowsHookEx**

CallWndProc 3.1

Syntax LRESULT CALLBACK CallWndProc(code, wParam, lParam)

The **CallWndProc** function is a library-defined callback function that the system calls whenever the **SendMessage** function is called. The system passes the message to the callback function before passing the message to the destination window procedure.

Parameters

code	Specifies whether the callback function should process the message or call the **CallNextHookEx** function. If the *code* parameter is less than zero, the callback function should

pass the message to **CallNextHookEx** without further processing.

wParam Specifies whether the message is sent by the current task. This parameter is nonzero if the message is sent; otherwise, it is NULL.

lParam Points to a structure that contains details about the message. The following shows the order, type, and description of each member of the structure:

Member	Description
lParam	Contains the *lParam* parameter of the message.
wParam	Contains the *wParam* parameter of the message.
uMsg	Specifies the message.
hWnd	Identifies the window that will receive the message.

Return Value The callback function should return zero.

Comments The **CallWndProc** callback function can examine or modify the message as necessary. Once the function returns control to the system, the message, with any modifications, is passed on to the window procedure.

This callback function must be in a dynamic-link library.

An application must install the callback function by specifying the WH_CALLWNDPROC filter type and the procedure-instance address of the callback function in a call to the **SetWindowsHookEx** function.

CallWndProc is a placeholder for the library-defined function name. The actual name must be exported by including it in an **EXPORTS** statement in the library's module-definition file.

See Also **CallNextHookEx, SendMessage, SetWindowsHookEx**

CBTProc 3.1

Syntax LRESULT CALLBACK CBTProc(code, wParam, lParam)

The **CBTProc** function is a library-defined callback function that the system calls before activating, creating, destroying, minimizing, maximizing, moving, or sizing a window; before completing a system command; before removing a mouse or keyboard event from the system message queue; before setting the input focus; or before synchronizing with the system message queue.

The value returned by the callback function determines whether to allow or prevent one of these operations.

Parameters

code — Specifies a computer-based-training (CBT) hook code that identifies the operation about to be carried out, or a value less than zero if the callback function should pass the *code*, *wParam*, and *lParam* parameters to the **CallNextHookEx** function. The *code* parameter can be one of the following:

Code	Meaning
HCBT_ACTIVATE	Indicates that the system is about to activate a window.
HCBT_CLICKSKIPPED	Indicates that the system has removed a mouse message from the system message queue. A CBT application that must install a journaling playback filter in response to the mouse message should do so when it receives this hook code.
HCBT_CREATEWND	Indicates that a window is about to be created. The system calls the callback function before sending the WM_CREATE or WM_NCCREATE message to the window. If the callback function returns TRUE, the system destroys the window—the **CreateWindow** function returns NULL, but the WM_DESTROY message is not sent to the window. If the callback function returns FALSE, the window is created normally. At the time of the HCBT_CREATEWND notification, the window has been created, but its final size and position may not have been determined, nor has its parent window been established. It is possible to send messages to the newly created window, although the window has not yet received WM_NCCREATE or WM_CREATE messages. It is possible to change the Z-order of the newly created window by modifying the **hwndInsertAfter** member of the **CBT_CREATEWND** structure.
HCBT_DESTROYWND	Indicates that a window is about to be destroyed.
HCBT_KEYSKIPPED	Indicates that the system has removed a keyboard message from the system message queue. A CBT application that must install a journaling playback filter in response to the keyboard message should do so when it receives this hook code.
HCBT_MINMAX	Indicates that a window is about to be minimized or maximized.
HCBT_MOVESIZE	Indicates that a window is about to be moved or sized.

Code	Meaning
HCBT_QS	Indicates that the system has retrieved a WM_QUEUESYNC message from the system message queue.
HCBT_SETFOCUS	Indicates that a window is about to receive the input focus.
HCBT_SYSCOMMAND	Indicates that a system command is about to be carried out. This allows a CBT application to prevent task switching by hot keys.

wParam This parameter depends on the *code* parameter. See the following Comments section for details.

lParam This parameter depends on the *code* parameter. See the following Comments section for details.

Return Value For operations corresponding to the following CBT hook codes, the callback function should return zero to allow the operation, or 1 to prevent it:

HCBT_ACTIVATE
HCBT_CREATEWND
HCBT_DESTROYWND
HCBT_MINMAX
HCBT_MOVESIZE
HCBT_SYSCOMMAND

The return value is ignored for operations corresponding to the following CBT hook codes:

HCBT_CLICKSKIPPED
HCBT_KEYSKIPPED
HCBT_QS

Comments The callback function should not install a playback hook except in the situations described in the preceding list of hook codes.

This callback function must be in a dynamic-link library.

An application must install the callback function by specifying the WH_CBT filter type and the procedure-instance address of the callback function in a call to the **SetWindowsHookEx** function.

CBTProc is a placeholder for the library-defined function name. The actual name must be exported by including it in an **EXPORTS** statement in the library's module-definition file.

The following table describes the *wParam* and *lParam* parameters for each HCBT_ constant.

Constant	*wParam*	*lParam*
HCBT_ACTIVATE	Specifies the handle of the window about to be activated.	Specifies a long pointer to a **CBTACTIVATESTRUCT** structure that contains the handle of the currently active window and specifies whether the activation is changing because of a mouse click.
HCBT_CLICKSKIPPED	Identifies the mouse message removed from the system message queue.	Specifies a long pointer to a **MOUSE-HOOKSTRUCT** structure that contains the hit-test code and the handle of the window for which the mouse message is intended. For a list of hit-test codes, see the description of the WM_NCHITTEST message.
HCBT_CREATEWND	Specifies the handle of the new window.	Specifies a long pointer to a **CBT_CREATEWND** data structure that contains initialization parameters for the window.
HCBT_DESTROYWND	Specifies the handle of the window about to be destroyed.	This parameter is undefined and should be set to 0L.
HCBT_KEYSKIPPED	Identifies the virtual key code.	Specifies the repeat count, scan code, key-transition code, previous key state, and context code. For more information, see the description of the WM_KEYUP or WM_KEYDOWN message.
HCBT_MINMAX	Specifies the handle of the window being minimized or maximized.	The low-order word specifies a show-window value (SW_) that specifies the operation. For a list of show-window values, see the description of the **ShowWindow** function. The high-order word is undefined.
HCBT_MOVESIZE	Specifies the handle of the window to be moved or sized.	Specifies a long pointer to a **RECT** structure that contains the coordinates of the window.
HCBT_QS	This parameter is undefined; it should be set to 0.	This parameter is undefined and should be set to 0L.
HCBT_SETFOCUS	Specifies the handle of the window gaining the input focus.	The low-order word specifies the handle of the window losing the input focus. The high-order word is undefined.

Constant	*wParam*	*lParam*
HCBT_SYSCOMMAND	Specifies a system-command value (SC_) that specifies the system command. For more information about system command values, see the description of the WM_SYSCOMMAND message.	If *wParam* is SC_HOTKEY, the low-order word of *lParam* contains the handle of the window that task switching will bring to the foreground. If *wParam* is not SC_HOTKEY and a System-menu command is chosen with the mouse, the low-order word of *lParam* contains the x-coordinate of the cursor and the high-order word contains the y-coordinate. If neither of these conditions is true, *lParam* is undefined.

See Also **CallNextHookEx, SetWindowsHookEx**

ChooseColor 3.1

Syntax #include <commdlg.h>
BOOL ChooseColor(lpcc)

function ChooseColor(var CC: TChooseColor): Bool;

The **ChooseColor** function creates a system-defined dialog box from which the user can select a color.

Parameters

lpcc Points to a **CHOOSECOLOR** structure that initially contains information necessary to initialize the dialog box. When the **ChooseColor** function returns, this structure contains information about the user's color selection. The **CHOOSECOLOR** structure has the following form:

```
#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;
```

Return Value The return value is nonzero if the function is successful. It is zero if an error occurs, if the user chooses the Cancel button, or if the user chooses the Close command on the System menu (often called the Control menu) to close the dialog box.

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

CDERR_FINDRESFAILURE
CDERR_INITIALIZATION
CDERR_LOCKRESFAILURE
CDERR_LOADRESFAILURE
CDERR_LOADSTRFAILURE
CDERR_MEMALLOCFAILURE
CDERR_MEMLOCKFAILURE
CDERR_NOHINSTANCE
CDERR_NOHOOK
CDERR_NOTEMPLATE
CDERR_STRUCTSIZE

Comments The dialog box does not support color palettes. The color choices offered by the dialog box are limited to the system colors and dithered versions of those colors.

If the hook function (to which the **lpfnHook** member of the **CHOOSECOLOR** structure points) processes the WM_CTLCOLOR message, this function must return a handle for the brush that should be used to paint the control background.

Example The following example initializes a **CHOOSECOLOR** structure and then creates a color-selection dialog box:

```
/* Color variables */

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

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

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

/* Initialize clr to black. */

clr = RGB(0, 0, 0);

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

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

/* Initialize the necessary CHOOSECOLOR members. */

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

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

ChooseFont 3.1

Syntax #include <commdlg.h>
BOOL ChooseFont(lpcf)

function ChooseFont(var ChooseFont: TChooseFont): Bool;

The **ChooseFont** function creates a system-defined dialog box from which the user can select a font, a font style (such as bold or italic), a point size, an effect (such as strikeout or underline), and a color.

Parameters *lpcf* Points to a **CHOOSEFONT** structure that initially contains information necessary to initialize the dialog box. When the **ChooseFont** function returns, this structure contains information about the user's font selection. The **CHOOSEFONT** structure has the following form:

```
#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;
```

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

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

CDERR_FINDRESFAILURE
CDERR_INITIALIZATION
CDERR_LOCKRESFAILURE
CDERR_LOADRESFAILURE
CDERR_LOADSTRFAILURE
CDERR_MEMALLOCFAILURE
CDERR_MEMLOCKFAILURE
CDERR_NOHINSTANCE
CDERR_NOHOOK
CDERR_NOTEMPLATE
CDERR_STRUCTSIZE
CFERR_MAXLESSTHANMIN
CFERR_NOFONTS

Example The following example initializes a **CHOOSEFONT** structure and then displays a font dialog box:

```
LOGFONT lf;
CHOOSEFONT cf;

/* Set all structure fields to zero. */

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

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

ChooseFont(&cf);
```

ClassFirst 3.1

Syntax #include <toolhelp.h>
BOOL ClassFirst(lpce)

function ClassFirst(lpClass: PClassEntry): Bool;

The **ClassFirst** function fills the specified structure with general information about the first class in the Windows class list.

Parameters

lpce — Points to a **CLASSENTRY** structure that will receive the class information. The **CLASSENTRY** structure has the following form:

```
#include <toolhelp.h>

typedef struct tagCLASSENTRY {   /* ce */
    DWORD   dwSize;
    HMODULE hInst;
    char    szClassName[MAX_CLASSNAME + 1];
    WORD    wNext;
}CLASSENTRY;
```

Return Value

The return value is nonzero if the function is successful. Otherwise, it is zero.

Comments

The **ClassFirst** function can be used to begin a walk through the Windows class list. To examine subsequent items in the class list, an application can use the **ClassNext** function.

Before calling **ClassFirst**, an application must initialize the **CLASSENTRY** structure and specify its size, in bytes, in the **dwSize** member. An application can examine subsequent entries in the Windows class list by using the **ClassNext** function.

For more specific information about an individual class, use the **GetClassInfo** function, specifying the name of the class and instance handle from the **CLASSENTRY** structure.

See Also

ClassNext, GetClassInfo

ClassNext — 3.1

Syntax

#include <toolhelp.h>
BOOL ClassNext(lpce)

function ClassNext(lpClass: PClassEntry): Bool;

The **ClassNext** function fills the specified structure with general information about the next class in the Windows class list.

Parameters *lpce* Points to a **CLASSENTRY** structure that will receive the class information. The **CLASSENTRY** structure has the following form:

```
#include <toolhelp.h>

typedef struct tagCLASSENTRY {  /* ce */
    DWORD   dwSize;
    HMODULE hInst;
    char    szClassName[MAX_CLASSNAME + 1];
    WORD    wNext;
}CLASSENTRY;
```

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

Comments The **ClassNext** function can be used to continue a walk through the Windows class list started by the **ClassFirst** function.

For more specific information about an individual class, use the **GetClassInfo** function with the name of the class and instance handle from the **CLASSENTRY** structure.

See Also **ClassFirst**

CloseDriver 3.1

Syntax LRESULT CloseDriver(hdrvr, lParam1, lParam2)

function CloseDriver(Driver: THandle; lParam1, lParam2: Longint): Longint;

The **CloseDriver** function closes an installable driver.

Parameters

hdrvr Identifies the installable driver to be closed. This parameter must have been obtained by a previous call to the **OpenDriver** function.

lParam1 Specifies driver-specific data.

lParam2 Specifies driver-specific data.

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

Comments When an application calls **CloseDriver** and the driver identified by *hdrvr* is the last instance of the driver, Windows calls the **DriverProc** function three times. On the first call, Windows sets the third **DriverProc** parameter, *wMessage*, to DRV_CLOSE; on the second call, Windows sets *wMessage* to DRV_DISABLE; and on the third call, Windows sets *wMessage* to DRV_FREE. When the driver identified by *hdrvr* is not the last instance of the driver, only DRV_CLOSE is sent. The values specified in the *lParam1* and *lParam2* parameters are passed to the *lParam1* and *lParam2* parameters of the **DriverProc** function.

See Also **DriverProc, OpenDriver**

CommDlgExtendedError 3.1

Syntax #include <commdlg.h>
DWORD CommDlgExtendedError(void)

function CommDlgExtendedError: Longint;

The **CommDlgExtendedError** function identifies the cause of the most recent error to have occurred during the execution of one of the following common dialog box procedures:

- **ChooseColor**
- **ChooseFont**
- **FindText**
- **GetFileTitle**
- **GetOpenFileName**
- **GetSaveFileName**
- **PrintDlg**
- **ReplaceText**

Parameters This function has no parameters.

Return Value The return value is zero if the prior call to a common dialog box procedure was successful. The return value is CDERR_DIALOGFAILURE if the dialog box could not be created. Otherwise, the return value is a nonzero integer that identifies an error condition.

Comments Following are the possible **CommDlgExtendedError** return values and the meaning of each:

Value	Meaning
CDERR_FINDRESFAILURE	Specifies that the common dialog box procedure failed to find a specified resource.
CDERR_INITIALIZATION	Specifies that the common dialog box procedure failed during initialization. This error often occurs when insufficient memory is available.
CDERR_LOADRESFAILURE	Specifies that the common dialog box procedure failed to load a specified resource.
CDERR_LOCKRESFAILURE	Specifies that the common dialog box procedure failed to lock a specified resource.
CDERR_LOADSTRFAILURE	Specifies that the common dialog box procedure failed to load a specified string.
CDERR_MEMALLOCFAILURE	Specifies that the common dialog box procedure was unable to allocate memory for internal structures.
CDERR_MEMLOCKFAILURE	Specifies that the common dialog box procedure was unable to lock the memory associated with a handle.
CDERR_NOHINSTANCE	Specifies that the ENABLETEMPLATE flag was set in the **Flags** member of a structure for the corresponding common dialog box but that the application failed to provide a corresponding instance handle.
CDERR_NOHOOK	Specifies that the ENABLEHOOK flag was set in the **Flags** member of a structure for the corresponding common dialog box but that the application failed to provide a pointer to a corresponding hook function.
CDERR_NOTEMPLATE	Specifies that the ENABLETEMPLATE flag was set in the **Flags** member of a structure for the corresponding common dialog box but that the application failed to provide a corresponding template.
CDERR_REGISTERMSGFAIL	Specifies that the **RegisterWindowMessage** function returned an error value when it was called by the common dialog box procedure.
CDERR_STRUCTSIZE	Specifies as invalid the **lStructSize** member of a structure for the corresponding common dialog box.
CFERR_NOFONTS	Specifies that no fonts exist.
CFERR_MAXLESSTHANMIN	Specifies that the maximum size given for the dialog box is less than the specified minimum size.

Value	Meaning
FNERR_BUFFERTOOSMALL	Specifies that the buffer for a filename is too small. (This buffer is pointed to by the **lpstrFile** member of the structure for a common dialog box.)
FNERR_INVALIDFILENAME	Specifies that a filename is invalid.
FNERR_SUBCLASSFAILURE	Specifies that an attempt to subclass a list box failed due to insufficient memory.
FRERR_BUFFERLENGTHZERO	Specifies that a member in a structure for the corresponding common dialog box points to an invalid buffer.
PDERR_CREATEICFAILURE	Specifies that the **PrintDlg** function failed when it attempted to create an information context.
PDERR_DEFAULTDIFFERENT	Specifies that an application has called the **PrintDlg** function with the DN_DEFAULTPRN flag set in the **wDefault** member of the **DEVNAMES** structure, but the printer described by the other structure members does not match the current default printer. (This happens when an application stores the **DEVNAMES** structure and the user changes the default printer by using Control Panel.) To use the printer described by the **DEVNAMES** structure, the application should clear the DN_DEFAULTPRN flag and call the **PrintDlg** function again. To use the default printer, the application should replace the **DEVNAMES** structure (and the **DEVMODE** structure, if one exists) with NULL; this selects the default printer automatically.
PDERR_DNDMMISMATCH	Specifies that the data in the **DEVMODE** and **DEVNAMES** structures describes two different printers.
PDERR_GETDEVMODEFAIL	Specifies that the printer driver failed to initialize a **DEVMODE** structure. (This error value applies only to printer drivers written for Windows versions 3.0 and later.)
PDERR_INITFAILURE	Specifies that the **PrintDlg** function failed during initialization.
PDERR_LOADDRVFAILURE	Specifies that the **PrintDlg** function failed to load the device driver for the specified printer.
PDERR_NODEFAULTPRN	Specifies that a default printer does not exist.
PDERR_NODEVICES	Specifies that no printer drivers were found.

Value	Meaning
PDERR_PARSEFAILURE	Specifies that the **PrintDlg** function failed to parse the strings in the [devices] section of the WIN.INI file.
PDERR_PRINTERNOTFOUND	Specifies that the [devices] section of the WIN.INI file did not contain an entry for the requested printer.
PDERR_RETDEFFAILURE	Specifies that the PD_RETURNDEFAULT flag was set in the **Flags** member of the **PRINTDLG** structure but that either the **hDevMode** or **hDevNames** member was nonzero.
PDERR_SETUPFAILURE	Specifies that the **PrintDlg** function failed to load the required resources.

See Also **ChooseColor, ChooseFont, FindText, GetFileTitle, GetOpenFileName, GetSaveFileName, PrintDlg, ReplaceText**

CopyCursor 3.1

Syntax HCURSOR CopyCursor(hinst, hcur)

function CopyCursor(hInst: THandle; hCur: HCursor): HCursor;

The **CopyCursor** function copies a cursor.

Parameters

hinst	Identifies the instance of the module that will copy the cursor.
hcur	Identifies the cursor to be copied.

Return Value The return value is the handle of the duplicate cursor if the function is successful. Otherwise, it is NULL.

Comments When it no longer requires a cursor, an application must destroy the cursor, using the **DestroyCursor** function.

The **CopyCursor** function allows an application or dynamic-link library to accept a cursor from another module. Because all resources are owned by the module in which they originate, a resource cannot be shared after the module is freed. **CopyCursor** allows an application to create a copy that the application then owns.

See Also **CopyIcon, DestroyCursor, GetCursor, SetCursor, ShowCursor**

CopyIcon 3.1

Syntax HICON CopyIcon(hinst, hicon)

function CopyIcon(hInst: THandle; Icon: HIcon): HIcon;

The **CopyIcon** function copies an icon.

Parameters

hinst	Identifies the instance of the module that will copy the icon.
hicon	Identifies the icon to be copied.

Return Value The return value is the handle of the duplicate icon if the function is successful. Otherwise, it is NULL.

Comments When it no longer requires an icon, an application should destroy the icon, using the **DestroyIcon** function.

The **CopyIcon** function allows an application or dynamic-link library to accept an icon from another module. Because all resources are owned by the module in which they originate, a resource cannot be shared after the module is freed. **CopyIcon** allows an application to create a copy that the application then owns.

See Also **CopyCursor, DestroyIcon, DrawIcon**

CopyLZFile 3.1

Syntax #include <lzexpand.h>
LONG CopyLZFile(hfSource, hfDest)

function CopyLZFile(Source, Dest: Integer): Longint;

The **CopyLZFile** function copies a source file to a destination file. If the source file is compressed, this function creates a decompressed destination file. If the source file is not compressed, this function duplicates the original file.

Parameters

hfSource	Identifies the source file.
hfDest	Identifies the destination file.

Return Value The return value specifies the size, in bytes, of the destination file if the function is successful. Otherwise, it is an error value less than zero; it may be one of the following:

Value	Meaning
LZERROR_BADINHANDLE	The handle identifying the source file was not valid.
LZERROR_BADOUTHANDLE	The handle identifying the destination file was not valid.
LZERROR_READ	The source file format was not valid.
LZERROR_WRITE	There is insufficient space for the output file.
LZERROR_GLOBALLOC	There is insufficient memory for the required buffers.
LZERROR_UNKNOWNALG	The file was compressed with an unrecognized compression algorithm.

Comments The **CopyLZFile** function is designed for copying or decompressing multiple files, or both. To allocate required buffers, an application should call the **LZStart** function prior to calling **CopyLZFile**. To free these buffers, an application should call the **LZDone** function after copying the files.

If the function is successful, the file identified by *hfDest* is decompressed.

If the source or destination file is opened by using a C run-time function (rather than by using the **_lopen** or **OpenFile** function), it must be opened in binary mode.

Example The following example uses the **CopyLZFile** function to create copies of four text files:

```
#define STRICT

#include <windows.h>
#include <lzexpand.h>

#define NUM_FILES   4

char *szSrc[NUM_FILES] =
    {"readme.txt", "data.txt", "update.txt", "list.txt"};
char*szDest[NUM_FILES]=
    {"readme.bak", "data.bak", "update.bak", "list.bak"};
OFSTRUCT ofStrSrc;
OFSTRUCT ofStrDest;
HFILE hfSrcFile, hfDstFile;
int i;

/* Allocate internal buffers for the CopyLZFile function. */

LZStart();

/* Open, copy, and then close the files. */

for (i = 0; i < NUM_FILES; i++) {
    hfSrcFile = LZOpenFile(szSrc[i], &ofStrSrc, OF_READ);
    hfDstFile = LZOpenFile(szDest[i], &ofStrDest, OF_CREATE);
```

```
        CopyLZFile(hfSrcFile, hfDstFile);
        LZClose(hfSrcFile);
        LZClose(hfDstFile);
    }

    LZDone(); /* free the internal buffers */
```

See Also **_lopen, LZCopy, LZDone, LZStart, OpenFile**

CPlApplet 3.1

Syntax LONG CALLBACK* CPlApplet(hwndCPl, iMessage, lParam1, lParam2)

TApplet_Proc = function(hWndCpl: HWnd; msg: Word; lParam1, lParam2: Longint): Longint;

The **CPlApplet** function serves as the entry point for a Control Panel dynamic-link library (DLL). This function is supplied by the application.

Parameters

hwndCPl	Identifies the main Control Panel window.
iMessage	Specifies the message being sent to the DLL.
lParam1	Specifies 32 bits of additional message-dependent information.
lParam2	Specifies 32 bits of additional message-dependent information.

Return Value The return value depends on the message.

Comments Use the *hwndCPl* parameter for dialog boxes or other windows that require a handle of a parent window.

CreateScalableFontResource 3.1

Syntax BOOL CreateScalableFontResource(fHidden, lpszResourceFile, lpszFontFile, lpszCurrentPath)

function CreateScalableFontResource(fHidden: HDC; lpszResourceFile, lpszFontFile, lpszCurrentPath: PChar): Bool;

The **CreateScalableFontResource** function creates a font resource file for the specified scalable font file.

Parameters

fHidden
Specifies whether the font is a read-only embedded font. This parameter can be one of the following values:

Value	Meaning
0	The font has read-write permission.
1	The font has read-only permission and should be hidden from other applications in the system. When this flag is set, the font is not enumerated by the **EnumFonts** or **EnumFontFamilies** function.

lpszResourceFile
Points to a null-terminated string specifying the name of the font resource file that this function creates.

lpszFontFile
Points to a null-terminated string specifying the scalable font file this function uses to create the font resource file. This parameter must specify either the filename and extension or a full path and filename, including drive and filename extension.

lpszCurrentPath
Points to a null-terminated string specifying either the path to the scalable font file specified in the *lpszFontFile* parameter or NULL, if *lpszFontFile* specifies a full path.

Return Value

The return value is nonzero if the function is successful. Otherwise, it is zero.

Comments

An application must use the **CreateScalableFontResource** function to create a font resource file before installing an embedded font. Font resource files for fonts with read-write permission should use the .FOT filename extension. Font resource files for read-only fonts should use a different extension (for example, .FOR) and should be hidden from other applications in the system by specifying 1 for the *fHidden* parameter. The font resource files can be installed by using the **AddFontResource** function.

When the *lpszFontFile* parameter specifies only a filename and extension, the *lpszCurrentPath* parameter must specify a path. When the *lpszFontFile* parameter specifies a full path, the *lpszCurrentPath* parameter must be NULL or a pointer to NULL.

When only a filename and extension is specified in the *lpszFontFile* parameter and a path is specified in the *lpszCurrentPath* parameter, the string in *lpszFontFile* is copied into the .FOT file as the .TTF file that belongs to this resource. When the **AddFontResource** function is called, the system assumes that the .TTF file has been copied into the SYSTEM directory (or into the main Windows directory in the case of a network installation). The .TTF file need not be in this directory when the **CreateScalableFontResource** function is called, because the *lpszCurrentPath* parameter contains the directory information. A resource created in this manner does not contain absolute path information and can be used in any Windows installation.

When a path is specified in the *lpszFontFile* parameter and NULL is specified in the *lpszCurrentPath* parameter, the string in *lpszFontFile* is copied into the .FOT file. In this case, when the **AddFontResource** function is called, the .TTF file must be at the location specified in the *lpszFontFile* parameter when the **CreateScalableFontResource** function was called; the *lpszCurrentPath* parameter is not needed. A resource created in this manner contains absolute references to paths and drives and will not work if the .TTF file is moved to a different location.

The **CreateScalableFontResource** function supports only TrueType scalable fonts.

Example The following example shows how to create a TrueType font file in the SYSTEM directory of the Windows startup directory:

```
CreateScalableFontResource(0, "c:\\windows\\system\\font.fot",
   "font.ttr", "c:\\windows\\system");

AddFontResource("c:\\windows\\system\\font.fot");
```

The following example shows how to create a TrueType font file in a specified directory:

```
CreateScalableFontResource(0, "c:\\windows\\system\\font.fot",
    "c:\\fontdir\\font.ttr", NULL);

AddFontResource("c:\\windows\\system\\font.fot");
```

The following example shows how to work with a standard embedded font:

```
HFONThfont;

/* Extract .TTF file into C:\MYDIR\FONT.TTR. */

CreateScalableFontResource(0,"font.fot","c:\\mydir\\font.ttr",NULL);

AddFontResource("font.fot");

hfont=CreateFont(...,CLIP_DEFAULT_PRECIS,...,"FONT");
    .
    . /* Use the font. */
    .
DeleteObject(hfont);

RemoveFontResource("font.fot");
    .
    . /* Delete C:\MYDIR\FONT.FOT and C:\MYDIR\FONT.TTR. */
    .
```

The following example shows how to work with a read-only embedded font:

```
HFONThfont;

/*Extract.TTF file into C:\MYDIR\FONT.TTR.*/

CreateScalableFontResource(1,"font.for","c:\\mydir\\font.ttr",NULL);

AddFontResource("font.for");

hfont=CreateFont(...,CLIP_EMBEDDED,...,"FONT");
    .
    . /* Use the font. */
    .
DeleteObject(hfont);

RemoveFontResource("font.for");
    .
    . /* Delete C:\MYDIR\FONT.FOR and C:\MYDIR\FONT.TTR. */
    .
```

See Also **AddFontResource**

DdeAbandonTransaction 3.1

Syntax #include <ddeml.h>
BOOL DdeAbandonTransaction(idInst, hConv, idTransaction)

function DdeAbandonTransaction(Inst: Longint; Conv: HConv; Transaction: Longint): Bool;

The **DdeAbandonTransaction** function abandons the specified asynchronous transaction and releases all resources associated with the transaction.

Parameters

idInst	Specifies the application-instance identifier obtained by a previous call to the **DdeInitialize** function.
hConv	Identifies the conversation in which the transaction was initiated. If this parameter is NULL, all transactions are abandoned (the *idTransaction* parameter is ignored).
idTransaction	Identifies the transaction to terminate. If this parameter is NULL, all active transactions in the specified conversation are abandoned.

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

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

DMLERR_DLL_NOT_INITIALIZED
DMLERR_INVALIDPARAMETER
DMLERR_NO_ERROR
DMLERR_UNFOUND_QUEUE_ID

Comments Only a dynamic data exchange (DDE) client application should call the **DdeAbandonTransaction** function. If the server application responds to the transaction after the client has called **DdeAbandonTransaction**, the system discards the transaction results. This function has no effect on synchronous transactions.

See Also **DdeClientTransaction, DdeGetLastError, DdeInitialize, DdeQueryConvInfo**

DdeAccessData 3.1

Syntax #include <ddeml.h>
BYTE FAR* DdeAccessData(hData, lpcbData)

function DdeAccessData(Data: HDDEData; DataSize: PLongint): Pointer;

The **DdeAccessData** function provides access to the data in the given global memory object. An application must call the **DdeUnaccessData** function when it is finished accessing the data in the object.

Parameters

hData	Identifies the global memory object to access.
lpcbData	Points to a variable that receives the size, in bytes, of the global memory object identified by the *hData* parameter. If this parameter is NULL, no size information is returned.

Return Value The return value points to the first byte of data in the global memory object if the function is successful. Otherwise, the return value is NULL.

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

DMLERR_DLL_NOT_INITIALIZED
DMLERR_INVALIDPARAMETER
DMLERR_NO_ERROR

Comments If the *hData* parameter has not been passed to a Dynamic Data Exchange Management Library (DDEML) function, an application can use the pointer returned by **DdeAccessData** for read-write access to the global memory object. If *hData* has already been passed to a DDEML function, the pointer can only be used for read-only access to the memory object.

Example The following example uses the **DdeAccessData** function to obtain a pointer to a global memory object, uses the pointer to copy data from the object to a local buffer, then frees the pointer:

```
HDDEDATA hData;
LPBYTE lpszAdviseData;
DWORD cbDataLen;
DWORD i;
char szData[128];

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

See Also **DdeAddData**, **DdeCreateDataHandle**, **DdeFreeDataHandle**, **DdeGetLastError**, **DdeUnaccessData**

DdeAddData 3.1

Syntax #include <ddeml.h>
HDDEDATA DdeAddData(hData, lpvSrcBuf, cbAddData, offObj)

function DdeAddData(Data: HDDEData; Src: Pointer; cb, Off: Longint): HDDEData;

The **DdeAddData** function adds data to the given global memory object. An application can add data beginning at any offset from the beginning of the object. If new data overlaps data already in the object, the new data overwrites the old data in the bytes where the overlap occurs. The contents of locations in the object that have not been written to are undefined.

Parameters

hData	Identifies the global memory object that receives additional data.
lpvSrcBuf	Points to a buffer containing the data to add to the global memory object.
cbAddData	Specifies the length, in bytes, of the data to be added to the global memory object.
offObj	Specifies an offset, in bytes, from the beginning of the global memory object. The additional data is copied to the object beginning at this offset.

Return Value The return value is a new handle of the global memory object if the function is successful. The new handle should be used in all references to the object. The return value is zero if an error occurs.

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

DMLERR_DLL_NOT_INITIALIZED
DMLERR_INVALIDPARAMETER
DMLERR_MEMORY_ERROR
DMLERR_NO_ERROR

Comments After a data handle has been used as a parameter in another Dynamic Data Exchange Management Library (DDEML) function or returned by a DDE callback function, the handle may only be used for read access to the global memory object identified by the handle.

If the amount of global memory originally allocated is not large enough to hold the added data, the **DdeAddData** function will reallocate a global memory object of the appropriate size.

Example The following example creates a global memory object, uses the **DdeAddData** function to add data to the object, and then passes the data to a client with an XTYP_POKE transaction:

```
DWORD idInst;           /* instance identifier     */
HDDEDATA hddeStrings;   /* data handle             */
HSZ hszMyItem;          /* item-name string handle */
DWORD offObj = 0;       /* offset in global object */
char szMyBuf[16];       /* temporary string buffer */
HCONV hconv;            /* conversation handle     */
DWORD dwResult;         /* transaction results     */
BOOL fAddAString;       /* TRUE if strings to add  */

/* Create a global memory object. */

hddeStrings=DdeCreateDataHandle(idInst,NULL,0,0,
    hszMyItem, CF_TEXT, 0);

/*
 * If a string is available, the application-defined function
 * IsThereAString() copies it to szMyBuf and returns TRUE. Otherwise,
 * it returns FALSE.
 */

while((fAddAString=IsThereAString())){

    /* Add the string to the global memory object. */

    DdeAddData(hddeStrings,            /* data handle      */
        &szMyBuf,                      /* string buffer    */
        (DWORD) strlen(szMyBuf) + 1,   /* character count  */
        offObj);                       /* offset in object */

    offObj = (DWORD) strlen(szMyBuf) + 1; /* adjust offset */
}

/* No more data to add, so poke it to the server. */

DdeClientTransaction((voidFAR*)hddeStrings,-1L,hconv,hszMyItem,
    CF_TEXT, XTYP_POKE, 1000, &dwResult);
```

See Also **DdeAccessData, DdeCreateDataHandle, DdeGetLastError, DdeUnaccessData**

DdeCallback 3.1

Syntax #include <ddeml.h>
HDDEDATA CALLBACK DdeCallback(type, fmt, hconv, hsz1, hsz2, hData, dwData1, dwData2)

TCallback = function(CallType, Fmt: Word; Conv: HConv; hsz1, hsz2: HSZ; Data: HDDEData; Data1, Data2: Longint): HDDEData;

The **DdeCallback** function is an application-defined dynamic data exchange (DDE) callback function that processes DDE transactions sent to the function as a result of DDE Management Library (DDEML) calls by other applications.

Parameters

type Specifies the type of the current transaction. This parameter consists of a combination of transaction-class flags and transaction-type flags. The following table describes each of the transaction classes and provides a list of the transaction types in each class.

Value	Meaning
XCLASS_BOOL	A DDE callback function should return TRUE or FALSE when it finishes processing a transaction that belongs to this class. Following are the XCLASS_BOOL transaction types: XTYP_ADVSTART XTYP_CONNECT
XCLASS_DATA	A DDE callback function should return a DDE data handle, CBR_BLOCK, or NULL when it finishes processing a transaction that belongs to this class. Following are the XCLASS_DATA transaction types: XTYP_ADVREQ XTYP_REQUEST XTYP_WILDCONNECT
XCLASS_FLAGS	A DDE callback function should return DDE_FACK, DDE_FBUSY, or DDE_FNOTPROCESSED when it finishes processing a transaction that belongs to this class. Following are the XCLASS_FLAGS transaction types: XTYP_ADVDATA XTYP_EXECUTE XTYP_POKE

Value	Meaning
XCLASS_NOTIFICATION	The transaction types that belong to this class are for notification purposes only. The return value from the callback function is ignored. Following are the XCLASS_NOTIFICATION transaction types: XTYP_ADVSTOP XTYP_CONNECT_CONFIRM XTYP_DISCONNECT XTYP_ERROR XTYP_MONITOR XTYP_REGISTER XTYP_XACT_COMPLETE XTYP_UNREGISTER

fmt	Specifies the format in which data is to be sent or received.
hconv	Identifies conversation associated with the current transaction.
hsz1	Identifies a string. The meaning of this parameter depends on the type of the current transaction. For more information, see the description of the transaction type.
hsz2	Identifies a string. The meaning of this parameter depends on the type of the current transaction. For more information, see the description of the transaction type.
hData	Identifies DDE data. The meaning of this parameter depends on the type of the current transaction. For more information, see the description of the transaction type.
dwData1	Specifies transaction-specific data. For more information, see the description of the transaction type.
dwData2	Specifies transaction-specific data. For more information, see the description of the transaction type.

Return Value The return value depends on the transaction class.

Comments The callback function is called asynchronously for transactions that do not involve creating or terminating conversations. An application that does not frequently accept incoming messages will have reduced DDE performance because DDEML uses messages to initiate transactions.

An application must register the callback function by specifying its address in a call to the **DdeInitialize** function. **DdeCallback** is a placeholder for the application- or library-defined function name. The actual name must be exported by including it in an **EXPORTS** statement in the application's module-definition file.

See Also **DdeEnableCallback, DdeInitialize**

DdeClientTransaction 3.1

Syntax #include <ddeml.h>
HDDEDATA DdeClientTransaction(lpvData, cbData, hConv, hszItem, uFmt, uType, uTimeout, lpuResult)

function DdeClientTransaction(Data: Pointer; DataLen: Longint; Conv: HConv; Item: HSZ; Fmt, DataType: Word; Timeout: Longint; Result: PLongint): HDDEData;

The **DdeClientTransaction** function begins a data transaction between a client and a server. Only a dynamic data exchange (DDE) client application can call this function, and only after establishing a conversation with the server.

Parameters

lpvData	Points to the beginning of the data that the client needs to pass to the server. Optionally, an application can specify the data handle (HDDEDATA) to pass to the server, in which case the *cbData* parameter should be set to –1. This parameter is required only if the *uType* parameter is XTYP_EXECUTE or XTYP_POKE. Otherwise, this parameter should be NULL.
cbData	Specifies the length, in bytes, of the data pointed to by the *lpvData* parameter. A value of –1 indicates that *lpvData* is a data handle that identifies the data being sent.
hConv	Identifies the conversation in which the transaction is to take place.
hszItem	Identifies the data item for which data is being exchanged during the transaction. This handle must have been created by a previous call to the **DdeCreateStringHandle** function. This parameter is ignored (and should be set to NULL) if the *uType* parameter is XTYP_EXECUTE.
uFmt	Specifies the standard clipboard format in which the data item is being submitted or requested.
uType	Specifies the transaction type. This parameter can be one of the following values:

Value	Meaning
XTYP_ADVSTART	Begins an advise loop. Any number of distinct advise loops can exist within a conversation. An application can alter the advise loop type by combining the XTYP_ADVSTART transaction type with one or more of the following flags: **Value** — **Meaning** XTYPF_NODATA — Instructs the server to notify the client of any data changes without actually sending the data. This flag gives the client the option of ignoring the notification or requesting the changed data from the server. XTYPF_ACKREQ — Instructs the server to wait until the client acknowledges that it received the previous data item before sending the next data item. This flag prevents a fast server from sending data faster than the client can process it.
XTYP_ADVSTOP	Ends an advise loop.
XTYP_EXECUTE	Begins an execute transaction.
XTYP_POKE	Begins a poke transaction.
XTYP_REQUEST	Begins a request transaction.

uTimeout Specifies the maximum length of time, in milliseconds, that the client will wait for a response from the server application in a synchronous transaction. This parameter should be set to TIMEOUT_ASYNC for asynchronous transactions.

lpuResult Points to a variable that receives the result of the transaction. An application that does not check the result can set this value to NULL. For synchronous transactions, the low-order word of this variable will contain any applicable DDE_ flags resulting from the transaction. This provides support for applications dependent on DDE_APPSTATUS bits. (It is recommended that applications no longer use these bits because they may not be supported in future versions of the DDE Management Library.) For asynchronous transactions, this variable is filled with a unique transaction identifier for use with the

DdeAbandonTransaction function and the XTYP_XACT_COMPLETE transaction.

Return Value The return value is a data handle that identifies the data for successful synchronous transactions in which the client expects data from the server. The return value is TRUE for successful asynchronous transactions and for synchronous transactions in which the client does not expect data. The return value is FALSE for all unsuccessful transactions.

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

DMLERR_ADVACKTIMEOUT
DMLERR_BUSY
DMLERR_DATAACKTIMEOUT
DMLERR_DLL_NOT_INITIALIZED
DMLERR_EXECACKTIMEOUT
DMLERR_INVALIDPARAMETER
DMLERR_MEMORY_ERROR
DMLERR_NO_CONV_ESTABLISHED
DMLERR_NO_ERROR
DMLERR_NOTPROCESSED
DMLERR_POKEACKTIMEOUT
DMLERR_POSTMSG_FAILED
DMLERR_REENTRANCY
DMLERR_SERVER_DIED
DMLERR_UNADVACKTIMEOUT

Comments When the application is finished using the data handle returned by the **DdeClientTransaction** function, the application should free the handle by calling the **DdeFreeDataHandle** function.

Transactions can be synchronous or asynchronous. During a synchronous transaction, the **DdeClientTransaction** function does not return until the transaction completes successfully or fails. Synchronous transactions cause the client to enter a modal loop while waiting for various asynchronous events. Because of this, the client application can still respond to user input while waiting on a synchronous transaction but cannot begin a second synchronous transaction because of the activity associated with the first. The **DdeClientTransaction** function fails if any instance of the same task has a synchronous transaction already in progress.

During an asynchronous transaction, the **DdeClientTransaction** function returns after the transaction is begun, passing a transaction identifier for reference. When the server's DDE callback function finishes processing an asynchronous transaction, the system sends an XTYP_XACT_COMPLETE transaction to the client. This transaction provides the client with the results of the asynchronous transaction that it initiated by calling the **DdeClientTransaction** function. A client application can choose to abandon an asynchronous transaction by calling the **DdeAbandonTransaction** function.

Example The following example requests an advise loop with a DDE server application:

```
HCONV hconv;
HSZ hszNow;
HDDEDATA hData;
DWORD dwResult;

hData = DdeClientTransaction(
    (LPBYTE) NULL, /* pass no data to server */
    0,             /* no data                */
    hconv,         /* conversation handle    */
    hszNow,        /* item name              */
    CF_TEXT,       /* clipboard format       */
    XTYP_ADVSTART, /* start an advise loop   */
    1000,          /* time-out in one second */
    &dwResult);    /* points to result flags */
```

See Also **DdeAbandonTransaction, DdeAccessData, DdeConnect, DdeConnectList, DdeCreateStringHandle**

DdeCmpStringHandles 3.1

Syntax #include <ddeml.h>
int DdeCmpStringHandles(hsz1, hsz2)

function DdeCmpStringHandles(hsz1, hsz2: HSZ): Integer;

The **DdeCmpStringHandles** function compares the values of two string handles. The value of a string handle is not related to the case of the associated string.

Parameters

hsz1	Specifies the first string handle.
hsz2	Specifies the second string handle.

Return Value The return value can be one of the following:

Value	Meaning
–1	The value of *hsz1* is either 0 or less than the value of *hsz2*.
0	The values of *hsz1* and *hsz2* are equal (both can be 0).
1	The value of *hsz2* is either 0 or less than the value of *hsz1*.

Comments An application that needs to do a case-sensitive comparison of two string handles should compare the string handles directly. An application should use **DdeCompStringHandles** for all other comparisons to preserve the case-sensitive nature of dynamic data exchange (DDE).

The **DdeCompStringHandles** function cannot be used to sort string handles alphabetically.

Example This example compares two service-name string handles and, if the handles are the same, requests a conversation with the server, then issues an XTYP_ADVSTART transaction:

```
HSZ hszClock;   /* service name */
HSZ hszTime;    /* topic name   */
HSZ hsz1;       /* unknown server          */
HCONV hConv;    /* conversation handle     */
DWORD dwResult; /* result flags            */
DWORD idInst;   /* instance identifier     */

/*
 * Compare unknown service name handle with the string handle
 * for the clock application.
 */

if(!DdeCmpStringHandles(hsz1,hszClock)){

    /*
     * If this is the clock application, start a conversation
     * with it and request an advise loop.
     */

    hConv = DdeConnect(idInst, hszClock, hszTime, NULL);
    if (hConv != (HCONV) NULL)
        DdeClientTransaction(NULL, 0, hConv, hszNow,
            CF_TEXT, XTYP_ADVSTART, 1000, &dwResult);

}
```

See Also **DdeAccessData, DdeCreateStringHandle, DdeFreeStringHandle**

DdeConnect 3.1

Syntax #include <ddeml.h>
HCONV DdeConnect(idInst, hszService, hszTopic, pCC)

function DdeConnect(Inst: Longint; Service, Topic: HSZ; CC: PConvContext): HConv;

The **DdeConnect** function establishes a conversation with a server application that supports the specified service name and topic name pair. If more than one such server exists, the system selects only one.

Parameters

idInst Specifies the application-instance identifier obtained by a previous call to the **DdeInitialize** function.

hszService Identifies the string that specifies the service name of the server application with which a conversation is to be established. This handle must have been created by a previous call to the **DdeCreateStringHandle** function. If this parameter is NULL, a conversation will be established with any available server.

hszTopic Identifies the string that specifies the name of the topic on which a conversation is to be established. This handle must have been created by a previous call to the **DdeCreateStringHandle** function. If this parameter is NULL, a conversation on any topic supported by the selected server will be established.

pCC Points to the **CONVCONTEXT** structure that contains conversation-context information. If this parameter is NULL, the server receives the default **CONVCONTEXT** structure during the XTYP_CONNECT or XTYP_WILDCONNECT transaction.

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

```
#include<ddeml.h>

typedef struct tagCONVCONTEXT { /* cc
*/
    UINT        cb;
    UINT        wFlags;
    UINT        wCountryID;
    int         iCodePage;
    DWORD       dwLangID;
    DWORD       dwSecurity;
}CONVCONTEXT;
```

Return Value The return value is the handle of the established conversation if the function is successful. Otherwise, it is NULL.

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

DMLERR_DLL_NOT_INITIALIZED
DMLERR_INVALIDPARAMETER
DMLERR_NO_CONV_ESTABLISHED
DMLERR_NO_ERROR

Comments The client application should not make assumptions regarding which server will be selected. If an instance-specific name is specified in the *hszService* parameter, a conversation will be established only with the specified instance. Instance-specific service names are passed to an application's dynamic data exchange callback function during the XTYP_REGISTER and XTYP_UNREGISTER transactions.

All members of the default **CONVCONTEXT** structure are set to zero except **cb**, which specifies the size of the structure, and **iCodePage**, which specifies CP_WINANSI (the default code page).

Example The following example creates a service-name string handle and a topic-name string handle, then attempts to establish a conversation with a server that supports the service name and topic name. If the attempt fails, the example retrieves an error value identifying the reason for the failure.

```
DWORD idInst = 0L;
HSZ hszClock;
HSZ hszTime;
HCONV hconv;
UINT uError;

hszClock = DdeCreateStringHandle(idInst, "Clock", CP_WINANSI);
hszTime = DdeCreateStringHandle(idInst, "Time", CP_WINANSI);

if ((hconv = DdeConnect(
    idInst,                 /* instance identifier            */
    hszClock,               /* server's service name          */
    hszTime,                /* topic name                     */
    NULL)) == NULL) {       /* use default CONVCONTEXT        */
    uError = DdeGetLastError(idInst);
}
```

See Also **DdeConnectList, DdeCreateStringHandle, DdeDisconnect, DdeDisconnectList, DdeInitialize**

DdeConnectList 3.1

Syntax #include <ddeml.h>
HCONVLIST DdeConnectList(idInst, hszService, hszTopic, hConvList, pCC)

function DdeConnectList(Inst: Longint; Service, Topic: HSZ; convList: HConvList; CC: PConvContext): HConvList;

The **DdeConnectList** function establishes a conversation with all server applications that support the specified service/topic name pair. An application can also use this function to enumerate a list of conversation handles by passing the function an existing conversation handle. During enumeration, the Dynamic Data Exchange Management Library (DDEML) removes the handles of any terminated conversations from the conversation list. The resulting conversation list contains the handles of all conversations currently established that support the specified service name and topic name.

Parameters

idInst Specifies the application-instance identifier obtained by a previous call to the **DdeInitialize** function.

hszService Identifies the string that specifies the service name of the server application with which a conversation is to be established. If this parameter is NULL, the system will attempt to establish conversations with all available servers that support the specified topic name.

hszTopic Identifies the string that specifies the name of the topic on which a conversation is to be established. This handle must have been created by a previous call to the **DdeCreateStringHandle** function. If this parameter is NULL, the system will attempt to establish conversations on all topics supported by the selected server (or servers).

hConvList Identifies the conversation list to be enumerated. This parameter should be set to NULL if a new conversation list is to be established.

pCC Points to the **CONVCONTEXT** structure that contains conversation-context information. If this parameter is NULL, the server receives the default **CONVCONTEXT** structure during the XTYP_CONNECT or XTYP_WILDCONNECT transaction.

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

```
#include <ddeml.h>

typedef struct tagCONVCONTEXT { /* cc
*/
    UINT        cb;
    UINT        wFlags;
    UINT        wCountryID;
    int         iCodePage;
    DWORD       dwLangID;
    DWORD       dwSecurity;
} CONVCONTEXT;
```

Return Value The return value is the handle of a new conversation list if the function is successful. Otherwise, it is NULL. The handle of the old conversation list is no longer valid.

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

DMLERR_DLL_NOT_INITIALIZED
DMLERR_INVALID_PARAMETER
DMLERR_NO_CONV_ESTABLISHED
DMLERR_NO_ERROR
DMLERR_SYS_ERROR

Comments An application must free the conversation-list handle returned by this function, regardless of whether any conversation handles within the list are active. To free the handle, an application can call the **DdeDisconnectList** function.

All members of the default **CONVCONTEXT** structure are set to zero except **cb**, which specifies the size of the structure, and **iCodePage**, which specifies CP_WINANSI (the default code page).

Example The following example uses the **DdeConnectList** function to establish a conversation with all servers that support the System topic, counts the servers, allocates a buffer for storing the server's service-name string handles, and then copies the handles to the buffer:

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

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

hconvList=DdeConnectList(idInst, (HSZ)NULL,hszSystem,
    (HCONV) NULL, (LPVOID) NULL);

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

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

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

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

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

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

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

/* Free the memory and terminate conversations. */

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

See Also **DdeConnect, DdeCreateStringHandle, DdeDisconnect, DdeDisconnectList, DdeInitialize, DdeQueryNextServer**

DdeCreateDataHandle 3.1

Syntax #include <ddeml.h>
HDDEDATA DdeCreateDataHandle(idInst, lpvSrcBuf, cbInitData, offSrcBuf, hszItem, uFmt, afCmd)

function DdeCreateDataHandle(Inst: Longint; Src: Pointer; cb, Off: Longint; Item: HSZ; Fmt, Cmd: Word): HDDEData;

The **DdeCreateDataHandle** function creates a global memory object and fills the object with the data pointed to by the *lpvSrcBuf* parameter. A dynamic data exchange (DDE) application uses this function during transactions that involve passing data to the partner application.

Parameters	*idInst*	Specifies the application-instance identifier obtained by a previous call to the **DdeInitialize** function.
	lpvSrcBuf	Points to a buffer that contains data to be copied to the global memory object. If this parameter is NULL, no data is copied to the object.
	cbInitData	Specifies the amount, in bytes, of memory to allocate for the global memory object. If this parameter is zero, the *lpvSrcBuf* parameter is ignored.
	offSrcBuf	Specifies an offset, in bytes, from the beginning of the buffer pointed to by the *lpvSrcBuf* parameter. The data beginning at this offset is copied from the buffer to the global memory object.
	hszItem	Identifies the string that specifies the data item corresponding to the global memory object. This handle must have been created by a previous call to the **DdeCreateStringHandle** function. If the data handle is to be used in an XTYP_EXECUTE transaction, this parameter must be set to NULL.
	uFmt	Specifies the standard clipboard format of the data.
	afCmd	Specifies the creation flags. This parameter can be HDATA_APPOWNED, which specifies that the server application that calls the **DdeCreateDataHandle** function will own the data handle that this function creates. This makes it possible for the server to share the data handle with multiple clients instead of creating a separate handle for each request. If this flag is set, the server must eventually free the shared memory object associated with this handle by using the **DdeFreeDataHandle** function. If this flag is not set, after the data handle is returned by the server's DDE callback function or used as a parameter in another DDE Management Library function, the handle becomes invalid in the application that creates the handle.

Return Value The return value is a data handle if the function is successful. Otherwise, it is NULL.

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

DMLERR_DLL_NOT_INITIALIZED
DMLERR_INVALIDPARAMETER
DMLERR_MEMORY_ERROR
DMLERR_NO_ERROR

Comments Any locations in the global memory object that are not filled are undefined.

After a data handle has been used as a parameter in another DDEML function or has been returned by a DDE callback function, the handle may be used only for read access to the global memory object identified by the handle.

If the application will be adding data to the global memory object (using the **DdeAddData** function) so that the object exceeds 64K in length, then the application should specify a total length (*cbInitData* + *offSrcData*) that is equal to the anticipated maximum length of the object. This avoids unnecessary data copying and memory reallocation by the system.

Example The following example processes the XTYP_WILDCONNECT transaction by returning a data handle to an array of **HSZPAIR** structures—one for each topic name supported:

```
#define CTOPICS 2

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

if (type == XTYP_WILDCONNECT) {

     /*
      * Scan the topic list and create array of HSZPAIR data
      * structures.
      */
    j = 0;
    for (i = 0; i < CTOPICS; i++) {
        if (hszTopic == (HSZ) NULL ||
                hszTopic == ahszTopicList[i]) {
            ahp[j].hszSvc = hszServ;
            ahp[j++].hszTopic = ahszTopicList[i];
        }
    }

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

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

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

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

See Also **DdeAccessData, DdeFreeDataHandle, DdeGetData, DdeInitialize**

DdeCreateStringHandle 3.1

Syntax #include <ddeml.h>
HSZ DdeCreateStringHandle(idInst, lpszString, codepage)

function DdeCreateStringHandle(Inst: Longint; psz: PChar; CodePage: Integer): HSZ;

The **DdeCreateStringHandle** function creates a handle that identifies the string pointed to by the *lpszString* parameter. A dynamic data exchange (DDE) client or server application can pass the string handle as a parameter to other DDE Management Library functions.

Parameters

idInst	Specifies the application-instance identifier obtained by a previous call to the **DdeInitialize** function.
lpszString	Points to a buffer that contains the null-terminated string for which a handle is to be created. This string may be any length.
codepage	Specifies the code page used to render the string. This value should be either CP_WINANSI or the value returned by the **GetKBCodePage** function. A value of zero implies CP_WINANSI.

Return Value The return value is a string handle if the function is successful. Otherwise, it is NULL.

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

DMLERR_INVALIDPARAMETER
DMLERR_NO_ERROR
DMLERR_SYS_ERROR

Comments Two identical strings always correspond to the same string handle. String handles are unique across all tasks that use the DDEML. That is, when an application creates a handle for a string and another application creates a handle for an identical string, the string handles returned to both applications are identical—regardless of case.

The value of a string handle is not related to the case of the string it identifies.

When an application has either created a string handle or received one in the callback function and has used the **DdeKeepStringHandle** function to keep it, the application must free that string handle when it is no longer needed.

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

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

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

Example The following example creates a service-name string handle and a topic-name string handle and then attempts to establish a conversation with a server that supports the service name and topic name. If the attempt fails, the example obtains an error value identifying the reason for the failure.

```
DWORD idInst = 0L;
HSZ hszClock;
HSZ hszTime;
HCONV hconv;
UINT uError;

hszClock = DdeCreateStringHandle(idInst, "Clock", CP_WINANSI);
hszTime = DdeCreateStringHandle(idInst, "Time", CP_WINANSI);

if ((hconv = DdeConnect(
    idInst,                 /* instance identifier          */
    hszClock,               /* server's service name        */
    hszTime,                /* topic name                   */
    NULL)) == NULL) {       /* use default CONVCONTEXT      */

    uError = DdeGetLastError(idInst);
}
```

See Also **DdeAccessData, DdeCmpStringHandles, DdeFreeStringHandle, DdeInitialize, DdeKeepStringHandle, DdeQueryString**

DdeDisconnect 3.1

Syntax

```
#include &lt;ddeml.h&gt;
BOOL DdeDisconnect(hConv)
```

```
function DdeDisconnect(Conv: HConv): Bool;
```

The **DdeDisconnect** function terminates a conversation started by either the **DdeConnect** or **DdeConnectList** function and invalidates the given conversation handle.

Parameters *hConv* Identifies the active conversation to be terminated.

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

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

DMLERR_DLL_NOT_INITIALIZED
DMLERR_NO_CONV_ESTABLISHED
DMLERR_NO_ERROR

Comments Any incomplete transactions started before calling **DdeDisconnect** are immediately abandoned. The XTYP_DISCONNECT transaction type is sent to the dynamic data exchange (DDE) callback function of the partner in the conversation. Generally, only client applications need to terminate conversations.

See Also **DdeConnect, DdeConnectList, DdeDisconnectList**

DdeDisconnectList 3.1

Syntax

```
#include &lt;ddeml.h&gt;
BOOL DdeDisconnectList(hConvList)
```

```
function DdeDisconnectList(ConvList: HConvList): Bool;
```

The **DdeDisconnectList** function destroys the given conversation list and terminates all conversations associated with the list.

Parameters *hConvList* Identifies the conversation list. This handle must have been created by a previous call to the **DdeConnectList** function.

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

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

DMLERR_DLL_NOT_INITIALIZED
DMLERR_INVALIDPARAMETER
DMLERR_NO_ERROR

Comments An application can use the **DdeDisconnect** function to terminate individual conversations in the list.

See Also **DdeConnect, DdeConnectList, DdeDisconnect**

DdeEnableCallback 3.1

Syntax #include <ddeml.h>
BOOL DdeEnableCallback(idInst, hConv, uCmd)

function DdeEnableCallback(Inst: Longint; Conv: HConv; Cmd: Word): Bool;

The **DdeEnableCallback** function enables or disables transactions for a specific conversation or for all conversations that the calling application currently has established.

After disabling transactions for a conversation, the system places the transactions for that conversation in a transaction queue associated with the application. The application should reenable the conversation as soon as possible to avoid losing queued transactions.

Parameters

idInst Specifies the application-instance identifier obtained by a previous call to the **DdeInitialize** function.

hConv Identifies the conversation to enable or disable. If this parameter is NULL, the function affects all conversations.

uCmd Specifies the function code. This parameter can be one of the following values:

Value	Meaning
EC_ENABLEALL	Enables all transactions for the specified conversation.
EC_ENABLEONE	Enables one transaction for the specified conversation.
EC_DISABLE	Disables all blockable transactions for the specified conversation. A server application can disable the following transactions: XTYP_ADVSTART XTYP_ADVSTOP XTYP_EXECUTE XTYP_POKE XTYP_REQUEST A client application can disable the following transactions: XTYP_ADVDATA XTYP_XACT_COMPLETE

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

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

DMLERR_DLL_NOT_INITIALIZED
DMLERR_NO_ERROR
DMLERR_INVALIDPARAMETER

Comments An application can disable transactions for a specific conversation by returning CBR_BLOCK from its dynamic data exchange (DDE) callback function. When the conversation is reenabled by using the **DdeEnableCallback** function, the system generates the same transaction as was in process when the conversation was disabled.

See Also **DdeConnect, DdeConnectList, DdeDisconnect, DdeInitialize**

DdeFreeDataHandle 3.1

Syntax

```
#include <ddeml.h>
BOOL DdeFreeDataHandle(hData)
```

```
function DdeFreeDataHandle(Data: HDDEData): Bool;
```

The **DdeFreeDataHandle** function frees a global memory object and deletes the data handle associated with the object.

Parameters *hData* Identifies the global memory object to be freed. This handle must have been created by a previous call to the **DdeCreateDataHandle** function or returned by the **DdeClientTransaction** function.

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

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

DMLERR_INVALIDPARAMETER
DMLERR_NO_ERROR

Comments An application must call **DdeFreeDataHandle** under the following circumstances:

- To free a global memory object that the application allocated by calling the **DdeCreateDataHandle** function if the object's data handle was never passed by the application to another Dynamic Data Exchange Management Library (DDEML) function
- To free a global memory object that the application allocated by specifying the HDATA_APPOWNED flag in a call to the **DdeCreateDataHandle** function
- To free a global memory object whose handle the application received from the **DdeClientTransaction** function

The system automatically frees an unowned object when its handle is returned by a dynamic data exchange (DDE) callback function or used as a parameter in a DDEML function.

Example The following example creates a global memory object containing help information, then frees the object after passing the object's handle to the client application:

```
DWORD idInst;
HSZ hszItem;
HDDEDATA hDataHelp;

char szDdeHelp[] = "DDEML test server help:\r\n"\
    "\tThe 'Server' (service) and 'Test' (topic) names may change.\r\n"\
    "Items supported under the 'Test' topic are:\r\n"\
    "\tCount:\tThis value increments on each data change.\r\n"\
    "\tRand:\tThis value is changed after each data change. \r\n"\
    "\t\tIn Runaway mode, the above items change after a request.\r\n"\
    "\tHuge:\tThis is randomly generated text data >64k that the\r\n"\
    "\t\ttest client can verify. It is recalculated on each\r\n"\
    "\t\trequest. This also verifies huge data poked or executed\r\n"\
    "\t\tfrom the test client.\r\n"\
```

```
"\tHelp:\tThis help information.  This data is APPOWNED.\r\n";

/* Create global memory object containing help information. */

if (!hDataHelp) {
    hDataHelp = DdeCreateDataHandle(idInst, szDdeHelp,
        strlen(szDdeHelp) + 1, 0, hszItem, CF_TEXT, HDATA_APPOWNED);
}

.
. /* Pass help information to client application. */
.

/* Free the global memory object. */

if (hDataHelp)
    DdeFreeDataHandle(hDataHelp);
```

See Also **DdeAccessData, DdeCreateDataHandle**

DdeFreeStringHandle 3.1

Syntax

#include <ddeml.h>
BOOL DdeFreeStringHandle(idInst, hsz)

function DdeFreeStringHandle(Inst: Longint; HSZ: HSZ): Bool;

The **DdeFreeStringHandle** function frees a string handle in the calling application.

Parameters

idInst	Specifies the application-instance identifier obtained by a previous call to the **DdeInitialize** function.
hsz	Identifies the string handle to be freed. This handle must have been created by a previous call to the **DdeCreateStringHandle** function.

Return Value

The return value is nonzero if the function is successful. Otherwise, it is zero.

Comments

An application can free string handles that it creates with the **DdeCreateStringHandle** function but should not free those that the system passed to the application's dynamic data exchange (DDE) callback function or those returned in the **CONVINFO** structure by the **DdeQueryConvInfo** function.

Example The following example frees string handles during the XTYP_DISCONNECT transaction:

```
DWORD idInst = 0L;
HSZhszClock;
HSZhszTime;
HSZhszNow;
UINTtype;

if (type == XTYP_DISCONNECT) {

    DdeFreeStringHandle(idInst, hszClock);
    DdeFreeStringHandle(idInst, hszTime);
    DdeFreeStringHandle(idInst, hszNow);

    return (HDDEDATA) NULL;
}
```

See Also **DdeCmpStringHandles, DdeCreateStringHandle, DdeInitialize, DdeKeepStringHandle, DdeQueryString**

DdeGetData 3.1

Syntax #include <ddeml.h>
DWORD DdeGetData(hData, pDest, cbMax, offSrc)

function DdeGetData(Data: HDDEData; Dst: Pointer; Max, Off: Longint): Longint;

The **DdeGetData** function copies data from the given global memory object to the specified local buffer.

Parameters

hData	Identifies the global memory object that contains the data to copy.
pDest	Points to the buffer that receives the data. If this parameter is NULL, the **DdeGetData** function returns the amount, in bytes, of data that would be copied to the buffer.
cbMax	Specifies the maximum amount, in bytes, of data to copy to the buffer pointed to by the *pDest* parameter. Typically, this parameter specifies the length of the buffer pointed to by *pDest*.
offSrc	Specifies an offset within the global memory object. Data is copied from the object beginning at this offset.

Return Value If the *pDest* parameter points to a buffer, the return value is the size, in bytes, of the memory object associated with the data handle or the size specified in the *cbMax* parameter, whichever is lower.

If the *pDest* parameter is NULL, the return value is the size, in bytes, of the memory object associated with the data handle.

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

DMLERR_DLL_NOT_INITIALIZED
DMLERR_INVALID_HDDEDATA
DMLERR_INVALIDPARAMETER
DMLERR_NO_ERROR

Example The following example copies data from a global memory object to a local buffer and then fills the **TIME** structure with data from the buffer:

```
HDDEDATA hData;
char szBuf[32];

typedef struct {
    int hour;
    int minute;
    int second;
} TIME;

DdeGetData(hData, (LPBYTE) szBuf, 32L, 0L);
sscanf(szBuf, "%d:%d:%d", &nTime.hour, &nTime.minute,
    &nTime.second);
```

See Also **DdeAccessData, DdeCreateDataHandle, DdeFreeDataHandle**

DdeGetLastError 3.1

Syntax #include <ddeml.h>
UINT DdeGetLastError(idInst)

function DdeGetLastError(Inst: Longint): Word;

The **DdeGetLastError** function returns the most recent error value set by the failure of a Dynamic Data Exchange Management Library (DDEML) function and resets the error value to DMLERR_NO_ERROR.

Parameters *idInst* Specifies the application-instance identifier obtained by a previous call to the **DdeInitialize** function.

Return Value The return value is the last error value. Following are the possible DDEML error values:

Value	Meaning
DMLERR_ADVACKTIMEOUT	A request for a synchronous advise transaction has timed out.
DMLERR_BUSY	The response to the transaction caused the DDE_FBUSY bit to be set.
DMLERR_DATAACKTIMEOUT	A request for a synchronous data transaction has timed out.
DMLERR_DLL_NOT_INITIALIZED	A DDEML function was called without first calling the **DdeInitialize** function, or an invalid instance identifier was passed to a DDEML function.
DMLERR_DLL_USAGE	An application initialized as APPCLASS_MONITOR has attempted to perform a DDE transaction, or an application initialized as APPCMD_CLIENTONLY has attempted to perform server transactions.
DMLERR_EXECACKTIMEOUT	A request for a synchronous execute transaction has timed out.
DMLERR_INVALIDPARAMETER	A parameter failed to be validated by the DDEML. Some of the possible causes are as follows: ▪ The application used a data handle initialized with a different item-name handle than that required by the transaction. ▪ The application used a data handle that was initialized with a different clipboard data format than that required by the transaction. ▪ The application used a client-side conversation handle with a server-side function or vise versa. ▪ The application used a freed data handle or string handle. ▪ More than one instance of the application used the same object.
DMLERR_LOW_MEMORY	A DDEML application has created a prolonged race condition (where the server application outruns the client), causing large amounts of memory to be consumed.

Value	Meaning
DMLERR_MEMORY_ERROR	A memory allocation failed.
DMLERR_NO_CONV_ESTABLISHED	A client's attempt to establish a conversation has failed.
DMLERR_NOTPROCESSED	A transaction failed.
DMLERR_POKEACKTIMEOUT	A request for a synchronous poke transaction has timed out.
DMLERR_POSTMSG_FAILED	An internal call to the **PostMessage** function has failed.
DMLERR_REENTRANCY	An application instance with a synchronous transaction already in progress attempted to initiate another synchronous transaction, or the **DdeEnableCallback** function was called from within a DDEML callback function.
DMLERR_SERVER_DIED	A server-side transaction was attempted on a conversation that was terminated by the client, or the server terminated before completing a transaction.
DMLERR_SYS_ERROR	An internal error has occurred in the DDEML.
DMLERR_UNADVACKTIMEOUT	A request to end an advise transaction has timed out.
DMLERR_UNFOUND_QUEUE_ID	An invalid transaction identifier was passed to a DDEML function. Once the application has returned from an XTYP_XACT_COMPLETE callback, the transaction identifier for that callback is no longer valid.

Example The following example calls the **DdeGetLastError** function if the **DdeCreateDataHandle** function fails:

```
DWORD idInst;
HDDEDATA hddeMyData;
HSZPAIR ahszp[2];
HSZ hszClock, hszTime;

/* Create string handles. */

hszClock = DdeCreateStringHandle(idInst, (LPSTR) "Clock",
    CP_WINANSI);
hszTime = DdeCreateStringHandle(idInst, (LPSTR) "Time",
    CP_WINANSI);

/* Copy handles to an HSZPAIR structure. */

ahszp[0].hszSvc    = hszClock;
```

```
ahszp[0].hszTopic = hszTime;
ahszp[1].hszSvc   = (HSZ) NULL;
ahszp[1].hszTopic = (HSZ) NULL;

/* Create a global memory object. */

hddeMyData = DdeCreateDataHandle(idInst, ahszp,
        sizeof(ahszp), 0, NULL, CF_TEXT, 0);
    if (hddeMyData == NULL)
        /*
         * Pass error value to application-defined error handling
         * function.
         */

        HandleError(DdeGetLastError(idInst));
```

See Also **DdeInitialize**

DdeInitialize 3.1

Syntax #include <ddeml.h>
UINT DdeInitialize(lpidInst, pfnCallback, afCmd, uRes)

function DdeInitialize(var Inst: Longint; Callback: TCallback; Cmd, Res: Longint): Word;

The **DdeInitialize** function registers an application with the Dynamic Data Exchange Management Library (DDEML). An application must call this function before calling any other DDEML function.

Parameters

lpidInst Points to the application-instance identifier. At initialization, this parameter should point to 0L. If the function is successful, this parameter points to the instance identifier for the application. This value should be passed as the *idInst* parameter in all other DDEML functions that require it. If an application uses multiple instances of the DDEML dynamic link library, the application should provide a different callback function for each instance.

If *lpidInst* points to a nonzero value, this implies a reinitialization of the DDEML. In this case, *lpidInst* must point to a valid application-instance identifier.

pfnCallback Points to the application-defined DDE callback function. This function processes DDE transactions sent by the system. For more information, see the description of the **DdeCallback** callback function.

afCmd Specifies an array of APPCMD_ and CBF_ flags. The APPCMD_ flags provide special instructions to the **DdeInitialize** function. The CBF_ flags set filters that prevent specific types of transactions from reaching the callback function. Using these flags enhances the performance of a DDE application by eliminating unnecessary calls to the callback function.

This parameter can be a combination of the following flags:

Flag	Meaning
APPCLASS_MONITOR	Makes it possible for the application to monitor DDE activity in the system. This flag is for use by DDE monitoring applications. The application specifies the types of DDE activity to monitor by combining one or more monitor flags with the APPCLASS_MONITOR flag. For details, see the following Comments section.
APPCLASS_STANDARD	Registers the application as a standard (nonmonitoring) DDEML application.
APPCMD_CLIENTONLY	Prevents the application from becoming a server in a DDE conversation. The application can be only a client. This flag reduces resource consumption by the DDEML. It includes the functionality of the CBF_FAIL_ALLSVRXACTIONS flag.
APPCMD_FILTERINITS	Prevents the DDEML from sending XTYP_CONNECT and XTYP_WILDCONNECT transactions to the application until the application has created its string handles and registered its service names or has turned off filtering by a subsequent call to the **DdeNameService** or **DdeInitialize** function. This flag is always in effect when an application calls **DdeInitialize** for the first time, regardless of whether the application specifies this flag. On subsequent calls to **DdeInitialize,** not specifying this flag turns off the application's service-name filters; specifying this flag turns on the application's service-name filters.

Flag	Meaning
CBF_FAIL_ALLSVRXACTIONS	Prevents the callback function from receiving server transactions. The system will return DDE_FNOTPROCESSED to each client that sends a transaction to this application. This flag is equivalent to combining all CBF_FAIL_ flags.
CBF_FAIL_ADVISES	Prevents the callback function from receiving XTYP_ADVSTART and XTYP_ADVSTOP transactions. The system will return DDE_FNOTPROCESSED to each client that sends an XTYP_ADVSTART or XTYP_ADVSTOP transaction to the server.
CBF_FAIL_CONNECTIONS	Prevents the callback function from receiving XTYP_CONNECT and XTYP_WILDCONNECT transactions.
CBF_FAIL_EXECUTES	Prevents the callback function from receiving XTYP_EXECUTE transactions. The system will return DDE_FNOTPROCESSED to a client that sends an XTYP_EXECUTE transaction to the server.
CBF_FAIL_POKES	Prevents the callback function from receiving XTYP_POKE transactions. The system will return DDE_FNOTPROCESSED to a client that sends an XTYP_POKE transaction to the server.
CBF_FAIL_REQUESTS	Prevents the callback function from receiving XTYP_REQUEST transactions. The system will return DDE_FNOTPROCESSED to a client that sends an XTYP_REQUEST transaction to the server.
CBF_FAIL_SELFCONNECTIONS	Prevents the callback function from receiving XTYP_CONNECT transactions from the application's own instance. This prevents an application from establishing a DDE conversation with its own instance. An application should use this flag if it needs to communicate with other instances of itself but not with itself.
CBF_SKIP_ALLNOTIFICATIONS	Prevents the callback function from receiving any notifications. This flag is equivalent combining all CBF_SKIP_ flags.

Flag	Meaning
CBF_SKIP_CONNECT_CONFIRMS	Prevents the callback function from receiving XTYP_CONNECT_CONFIRM notifications.
CBF_SKIP_DISCONNECTS	Prevents the callback function from receiving XTYP_DISCONNECT notifications.
CBF_SKIP_REGISTRATIONS	Prevents the callback function from receiving XTYP_REGISTER notifications.
CBF_SKIP_UNREGISTRATIONS	Prevents the callback function from receiving XTYP_UNREGISTER notifications.

uRes Reserved; must be set to 0L.

Return Value The return value is one of the following:

DMLERR_DLL_USAGE
DMLERR_INVALIDPARAMETER
DMLERR_NO_ERROR
DMLERR_SYS_ERROR

Comments An application that uses multiple instances of the DDEML must not pass DDEML objects between instances.

A DDE monitoring application should not attempt to perform DDE (establish conversations, issue transactions, and so on) within the context of the same application instance.

A synchronous transaction will fail with a DMLERR_REENTRANCY error if any instance of the same task has a synchronous transaction already in progress.

A DDE monitoring application can combine one or more of the following monitor flags with the APPCLASS_MONITOR flag to specify the types of DDE activity to monitor:

Flag	Meaning
MF_CALLBACKS	Notifies the callback function whenever a transaction is sent to any DDE callback function in the system.
MF_CONV	Notifies the callback function whenever a conversation is established or terminated.
MF_ERRORS	Notifies the callback function whenever a DDE error occurs.

Flag	Meaning
MF_HSZ_INFO	Notifies the callback function whenever a DDE application creates, frees, or increments the use count of a string handle or whenever a string handle is freed as a result of a call to the **DdeUninitialize** function.
MF_LINKS	Notifies the callback function whenever an advise loop is started or ended.
MF_POSTMSGS	Notifies the callback function whenever the system or an application posts a DDE message.
MF_SENDMSGS	Notifies the callback function whenever the system or an application sends a DDE message.

Example The following example obtains a procedure-instance address for a DDE callback function, then initializes the application with the DDEML.

```
DWORD idInst = 0L;
FARPROC lpDdeProc;

lpDdeProc = MakeProcInstance((FARPROC) DDECallback, hInst);
if (DdeInitialize((LPDWORD) &idInst, (PFNCALLBACK) lpDdeProc,
        APPCMD_CLIENTONLY, 0L))
    return FALSE;
```

See Also **DdeClientTransaction, DdeConnect, DdeCreateDataHandle, DdeEnableCallback, DdeNameService, DdePostAdvise, DdeUninitialize**

DdeKeepStringHandle 3.1

Syntax #include <ddeml.h>
BOOL DdeKeepStringHandle(idInst, hsz)

function DdeKeepStringHandle(Inst: Longint; HSZ: HSZ): Bool;

The **DdeKeepStringHandle** function increments the usage count (increases it by one) associated with the given handle. This function makes it possible for an application to save a string handle that was passed to the application's dynamic data exchange (DDE) callback function. Otherwise, a string handle passed to the callback function is deleted when the callback function returns.

Parameters

idInst	Specifies the application-instance identifier obtained by a previous call to the **DdeInitialize** function.
hsz	Identifies the string handle to be saved.

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

Example The following example is a portion of a DDE callback function that increases the usage count and saves a local copy of two string handles:

```
HSZ hsz1;
HSZ hsz2;
static HSZ hszServerBase;
static HSZ hszServerInst;
DWORD idInst;

case XTYP_REGISTER:

    /* Keep the handles for later use. */

    DdeKeepStringHandle(idInst, hsz1);
    DdeKeepStringHandle(idInst, hsz2);
    hszServerBase = hsz1;
    hszServerInst = hsz2;
    .
    . /* Finish processing the transaction. */
    .
```

See Also **DdeCreateStringHandle, DdeFreeStringHandle, DdeInitialize, DdeQueryString**

DdeNameService 3.1

Syntax #include <ddeml.h>
HDDEDATA DdeNameService(idInst, hsz1, hszRes, afCmd)

function DdeNameService(Inst: Longint; hsz1, hsz2: HSZ; Cmd: Word): HDDEData;

The **DdeNameService** function registers or unregisters the service names that a dynamic data exchange (DDE) server supports. This function causes the system to send XTYP_REGISTER or XTYP_UNREGISTER transactions to other running DDE Management Library (DDEML) client applications.

A server application should call this function to register each service name that it supports and to unregister names that it previously registered but no longer supports. A server should also call this function to unregister its service names just before terminating.

Parameters

idInst Specifies the application-instance identifier obtained by a previous call to the **DdeInitialize** function.

hsz1	Identifies the string that specifies the service name that the server is registering or unregistering. An application that is unregistering all of its service names should set this parameter to NULL.
hszRes	Reserved; should be set to NULL.
afCmd	Specifies the service-name flags. This parameter can be one of the following values:

Value	Meaning
DNS_REGISTER	Registers the given service name.
DNS_UNREGISTER	Unregisters the given service name. If the *hsz1* parameter is NULL, all service names registered by the server will be unregistered.
DNS_FILTERON	Turns on service-name initiation filtering. This filter prevents a server from receiving XTYP_CONNECT transactions for service names that it has not registered. This is the default setting for this filter. If a server application does not register any service names, the application cannot receive XTYP_WILDCONNECT transactions.
DNS_FILTEROFF	Turns off service-name initiation filtering. If this flag is set, the server will receive an XTYP_CONNECT transaction whenever another DDE application calls the **DdeConnect** function, regardless of the service name.

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

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

DMLERR_DLL_NOT_INITIALIZED
DMLERR_DLL_USAGE
DMLERR_INVALIDPARAMETER
DMLERR_NO_ERROR

Comments The service name identified by the *hsz1* parameter should be a base name (that is, the name should contain no instance-specific information). The system generates an instance-specific name and sends it along with the base name during the XTYP_REGISTER and XTYP_UNREGISTER transactions. The receiving applications can then connect to the specific application instance.

Example The following example initializes an application with the DDEML, creates frequently used string handles, and registers the application's service name:

```
HSZ hszClock;
HSZ hszTime;
HSZ hszNow;
HINSTANCE hinst;
DWORD idInst = 0L;
FARPROC lpDdeProc;

/* Initialize the application for the DDEML. */

lpDdeProc = MakeProcInstance((FARPROC) DdeCallback, hinst);
if (!DdeInitialize((LPDWORD) &idInst, (PFNCALLBACK) lpDdeProc,
        APPCMD_FILTERINITS | CBF_FAIL_EXECUTES, 0L)) {

    /* Create frequently used string handles. */

    hszTime = DdeCreateStringHandle(idInst, "Time", CP_WINANSI);
    hszNow = DdeCreateStringHandle(idInst, "Now", CP_WINANSI);
    hszClock = DdeCreateStringHandle(idInst, "Clock", CP_WINANSI);

    /* Register the service name. */

    DdeNameService(idInst, hszClock, (HSZ) NULL, DNS_REGISTER);

}
```

See Also **DdeConnect, DdeConnectList, DdeInitialize**

DdePostAdvise 3.1

Syntax #include <ddeml.h>
BOOL DdePostAdvise(idInst, hszTopic, hszItem)

function DdePostAdvise(Inst: Longint; Topic, Item: HSZ): Bool;

The **DdePostAdvise** function causes the system to send an XTYP_ADVREQ transaction to the calling (server) application's dynamic data exchange (DDE) callback function for each client that has an advise loop active on the specified topic or item name pair. A server application should call this function whenever the data associated with the topic or item name pair changes.

Parameters

idInst	Specifies the application-instance identifier obtained by a previous call to the **DdeInitialize** function.
hszTopic	Identifies a string that specifies the topic name. To send notifications for all topics with active advise loops, an application can set this parameter to NULL.

hszItem	Identifies a string that specifies the item name. To send notifications for all items with active advise loops, an application can set this parameter to NULL.

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

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

DMLERR_DLL_NOT_INITIALIZED
DMLERR_DLL_USAGE
DMLERR_NO_ERROR

Comments A server that has nonenumerable topics or items should set the *hszTopic* and *hszItem* parameters to NULL so that the system will generate transactions for all active advise loops. The server's DDE callback function returns NULL for any advise loops that do not need to be updated.

If a server calls **DdePostAdvise** with a topic/item/format name set that includes the set currently being handled in a XTYP_ADVREQ callback, a stack overflow may result.

Example The following example calls the **DdePostAdvise** function whenever the time changes:

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

TIME tmTime;
DWORD idInst;
HSZ hszTime;
HSZ hszNow;
TIME tmCurTime;

    .
    . /* Fill tmCurTime with the current time. */
    .

/* Check for any change in second, minute, or hour. */

if ((tmCurTime.second != tmTime.second) ||
        (tmCurTime.minute != tmTime.minute) ||
        (tmCurTime.hour   != tmTime.hour)) {

    /* Send the current time to the clients. */

    DdePostAdvise(idInst, hszTime, hszNow);
```

See Also **DdeInitialize**

DdeQueryConvInfo 3.1

Syntax #include <ddeml.h>
UINT DdeQueryConvInfo(hConv, idTransaction, lpConvInfo)

function DdeQueryConvInfo(Conv: HConv; Transaction: Longint; ConvInfo: PConvInfo): Word;

The **DdeQueryConvInfo** function retrieves information about a dynamic data exchange (DDE) transaction and about the conversation in which the transaction takes place.

Parameters

hConv Identifies the conversation.

idTransaction Specifies the transaction. For asynchronous transactions, this parameter should be a transaction identifier returned by the **DdeClientTransaction** function. For synchronous transactions, this parameter should be QID_SYNC.

lpConvInfo Points to the **CONVINFO** structure that will receive information about the transaction and conversation. The **cb** member of the **CONVINFO** structure must specify the length of the buffer allocated for the structure.

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

```
#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;
```

Return Value The return value is the number of bytes copied into the **CONVINFO** structure, if the function is successful. Otherwise, it is zero.

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

DMLERR_DLL_NOT_INITIALIZED
DMLERR_NO_CONV_ESTABLISHED
DMLERR_NO_ERROR
DMLERR_UNFOUND_QUEUE_ID

Example The following example fills a **CONVINFO** structure with information about a synchronous conversation and then obtains the names of the partner application and topic:

```
DWORD idInst;
HCONV hConv;
CONVINFO ci;
WORD wError;
char szSvcPartner[32];
char szTopic[32];
DWORD cchServ, cchTopic;

if (!DdeQueryConvInfo(hConv, QID_SYNC, &ci))
    wError = DdeGetLastError(idInst);

else {
    cchServ = DdeQueryString(idInst, ci.hszSvcPartner,
        (LPSTR) &szSvcPartner, sizeof(szSvcPartner),
        CP_WINANSI);
    cchTopic = DdeQueryString(idInst, ci.hszTopic,
        (LPSTR) &szTopic, sizeof(szTopic),
        CP_WINANSI);
}
```

See Also **DdeConnect, DdeConnectList, DdeQueryNextServer**

DdeQueryNextServer 3.1

Syntax #include <ddeml.h>
HCONV DdeQueryNextServer(hConvList, hConvPrev)

function DdeQueryNextServer(ConvList: HConvList; ConvPrev: HConv): HConv;

The **DdeQueryNextServer** function obtains the next conversation handle in the given conversation list.

Parameters *hConvList* Identifies the conversation list. This handle must have been created by a previous call to the **DdeConnectList** function.

hConvPrev Identifies the conversation handle previously returned by this function. If this parameter is NULL, this function returns the first conversation handle in the list.

Return Value The return value is the next conversation handle in the list if the list contains any more conversation handles. Otherwise, it is NULL.

Example The following example uses the **DdeQueryNextServer** function to count the number of conversation handles in a conversation list and to copy the service-name string handles of the servers to a local buffer:

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

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

hconvList=DdeConnectList(idInst,(HSZ)NULL,hszSystem,
    (HCONV) NULL, (LPVOID) NULL);

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

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

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

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

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

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

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

/* Free the memory and terminate conversations. */

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

See Also **DdeConnectList, DdeDisconnectList**

DdeQueryString 3.1

Syntax

```
#include <ddeml.h>
DWORD DdeQueryString(idInst, hsz, lpsz, cchMax, codepage)
```

```
function DdeQueryString(Inst: Longint; HSZ: HSZ; psz: PChar; Max:
Longint; CodePage: Integer): Longint;
```

The **DdeQueryString** function copies text associated with a string handle into a buffer.

The string returned in the buffer is always null-terminated. If the string is longer than (*cchMax* – 1), only the first (*cchMax* – 1) characters of the string are copied.

If the *lpsz* parameter is NULL, this function obtains the length, in bytes, of the string associated with the string handle. The length does not include the terminating null character.

Parameters

idInst	Specifies the application-instance identifier obtained by a previous call to the **DdeInitialize** function.
hsz	Identifies the string to copy. This handle must have been created by a previous call to the **DdeCreateStringHandle** function.
lpsz	Points to a buffer that receives the string. To obtain the length of the string, this parameter should be set to NULL.
cchMax	Specifies the length, in bytes, of the buffer pointed to by the *lpsz* parameter. If the string is longer than (*cchMax* – 1), it will be truncated. If the *lpsz* parameter is set to NULL, this parameter is ignored.
codepage	Specifies the code page used to render the string. This value should be either CP_WINANSI or the value returned by the **GetKBCodePage** function.

Return Value The return value is the length, in bytes, of the returned text (not including the terminating null character) if the *lpsz* parameter specified a valid pointer. The return value is the length of the text associated with the *hsz* parameter (not including the terminating null character) if the *lpsz* parameter specified a NULL pointer. The return value is NULL if an error occurs.

Example The following example uses the **DdeQueryString** function to obtain a service name and topic name that a server has registered:

```
UINT type;

HSZ hsz1;
HSZ hsz2;
char szBaseName[16];
char szInstName[16];

if (type == XTYP_REGISTER) {

/* Copy the base service name to a buffer. */

    DdeQueryString(idInst, hsz1, (LPSTR) &szBaseName,
        sizeof(szBaseName), CP_WINANSI);

    /* Copy the instance-specific service name to a buffer. */

    DdeQueryString(idInst, hsz2, (LPSTR) &szInstName,
        sizeof(szInstName), CP_WINANSI);
    return (HDDEDATA) TRUE;
}
```

See Also **DdeCmpStringHandles, DdeCreateStringHandle, DdeFreeStringHandle, DdeInitialize**

DdeReconnect 3.1

Syntax #include <ddeml.h>
HCONV DdeReconnect(hConv)

function DdeReconnect(Conv: HConv): HConv;

The **DdeReconnect** function allows a client Dynamic Data Exchange Management Library (DDEML) application to attempt to reestablish a conversation with a service that has terminated a conversation with the client. When the conversation is reestablished, the DDEML attempts to reestablish any preexisting advise loops.

Parameters

hConv	Identifies the conversation to be reestablished. A client must have obtained the conversation handle by a previous call to the **DdeConnect** function.

Return Value The return value is the handle of the reestablished conversation if the function is successful. The return value is NULL if the function fails.

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

DMLERR_DLL_NOT_INITIALIZED
DMLERR_INVALIDPARAMETER
DMLERR_NO_CONV_ESTABLISHED
DMLERR_NO_ERROR

Example The following example shows the context within which an application should call the **DdeReconnect** function:

```
HDDEDATAEXPENTRYDdeCallback(wType,wFmt,hConv,hsz1,
    hsz2, hData, dwData1, dwData2)
WORD wType;        /* transaction type              */
WORD wFmt;         /* clipboard format              */
HCONV hConv;       /* handle of the conversation    */
HSZ hsz1;          /* handle of a string            */
HSZ hsz2;          /* handle of a string            */
HDDEDATA hData;    /* handle of a global memory object */
DWORD dwData1;     /* transaction-specific data     */
DWORD dwData2;     /* transaction-specific data     */
{
    BOOL fAutoReconnect;

    switch (wType) {
        case XTYP_DISCONNECT:
            if (fAutoReconnect) {
                DdeReconnect(hConv); /* attempt to reconnect */
            }
            return 0;

        .
        . /* Process other transactions. */
        .
    }
}
```

See Also **DdeConnect, DdeDisconnect**

DdeSetUserHandle 3.1

Syntax #include <ddeml.h>
BOOL DdeSetUserHandle(hConv, id, hUser)

function DdeSetUserHandle(Conv: HConv; ID, User: Longint): Bool;

The **DdeSetUserHandle** function associates an application-defined 32-bit value with a conversation handle and transaction identifier. This is useful for simplifying the processing of asynchronous transactions. An application can use the **DdeQueryConvInfo** function to retrieve this value.

Parameters	*hConv*	Identifies the conversation.
	id	Specifies the transaction identifier of an asynchronous transaction. An application should set this parameter to QID_SYNC if no asynchronous transaction is to be associated with the *hUser* parameter.
	hUser	Identifies the value to associate with the conversation handle.

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

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

DMLERR_DLL_NOT_INITIALIZED
DMLERR_INVALIDPARAMETER
DMLERR_NO_ERROR
DMLERR_UNFOUND_QUEUE_ID

See Also **DdeQueryConvInfo**

DdeUnaccessData 3.1

Syntax #include <ddeml.h>
BOOL DdeUnaccessData(hData)

function DdeUnaccessData(Data: HDDEData): Bool;

The **DdeUnaccessData** function frees a global memory object. An application must call this function when it is finished accessing the object.

Parameters	*hData*	Identifies the global memory object.

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

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

DMLERR_DLL_NOT_INITIALIZED
DMLERR_INVALIDPARAMETER
DMLERR_NO_ERROR

Example The following example obtains a pointer to a global memory object, uses the pointer to copy data from the object to a local buffer, and then uses the **DdeUnaccessData** function to free the object:

```
HDDEDATA hData;
LPBYTE lpszAdviseData;
DWORD cbDataLen;
DWORD i;
char szData[128];

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

See Also **DdeAccessData, DdeAddData, DdeCreateDataHandle, DdeFreeDataHandle**

DdeUninitialize 3.1

Syntax #include <ddeml.h>
BOOL DdeUninitialize(idInst)

function DdeUninitialize(Inst: Longint): Bool;

The **DdeUninitialize** function frees all Dynamic Data Exchange Management Library (DDEML) resources associated with the calling application.

Parameters

idInst	Specifies the application-instance identifier obtained by a previous call to the **DdeInitialize** function.

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

Comments The **DdeUninitialize** function terminates any conversations currently open for the application. If the partner in a conversation fails to terminate its end of the conversation, the system may enter a modal loop while it waits for the conversation to terminate. A timeout period is associated with this loop. If the timeout period expires before the conversation has terminated, a message box appears that gives the user the choice of waiting for another timeout period (Retry), waiting indefinitely (Ignore), or exiting the modal loop (Abort).

An application should wait until its windows are no longer visible and its message loop has terminated before calling this function.

See Also **DdeDisconnect, DdeDisconnectList, DdeInitialize**

DebugOutput 3.1

Syntax void FAR _cdecl DebugOutput(flags, lpszFmt, ...)

The **DebugOutput** function sends a message to the debugging terminal. Applications can apply the formatting codes to the message string and use filters and options to control the message category.

Parameters

flags Specifies the type of message to be sent to the debugging terminal. This parameter can be one of the following values:

Value	Meaning
DBF_TRACE	The message reports that no error has occurred and supplies information that may be useful during debugging. Example: "KERNEL: Loading SAMPLE.DLL"
DBF_WARNING	The message reports a situation that may or may not be an error, depending on the circumstances. Example: "KERNEL: LoadString failed"
DBF_ERROR	The message reports an error resulting from a failed call to a Windows function. The application continues to run. Example: "KERNEL: Invalid local heap"
DBF_FATAL	The message reports an error that will terminate the application. Example: "USER: Obsolete function SetDeskWallpaper called"

lpszFmt Points to a formatting string identical to the formatting strings used by the Windows function **wsprintf**. This string must be less than 160 characters long. Any additional formatting can be done by supplying additional parameters following *lpszFmt*.

... Specifies zero or more optional arguments. The number and type of arguments depends on the corresponding format-control character sequences specified in the *lpszFmt* parameter.

Return Value This function does not return a value.

Comments The messages sent by the **DebugOutput** function are affected by the system debugging options and trace-filter flags that are set and retrieved by using the **GetWinDebugInfo** and **SetWinDebugInfo** functions. These options and flags are stored in a **WINDEBUGINFO** structure.

Unlike most other Windows functions, **DebugOutput** uses the C calling convention (**_cdecl**), rather than the Pascal calling convention. As a result, the caller must pop arguments off the stack. Also, arguments must be pushed on the stack from right to left. In C-language modules, the C compiler performs this task.

Any application that uses this function must explicitly declare it as an import function. The following information must be included in the **IMPORTS** section of the application's module-definition file:

```
IMPORTS
    kernel._DebugOutput
```

See Also **GetWinDebugInfo, OutputDebugString, SetWinDebugInfo, wsprintf**

DebugProc 3.1

Syntax LRESULT CALLBACK DebugProc(code, wParam, lParam)

The **DebugProc** function is a library-defined callback function that the system calls before calling any other filter installed by the **SetWindowsHookEx** function. The system passes information about the filter about to be called to the **DebugProc** callback function. The callback function can examine the information and determine whether to allow the filter to be called.

Parameters

code Specifies the hook code. Currently, HC_ACTION is the only positive valid value. If this parameter is less than zero, the callback function must call the **CallNextHookEx** function without any further processing.

wParam Specifies the task handle of the task that installed the filter about to be called.

lParam Contains a long pointer to a **DEBUGHOOKINFO** structure. The **DEBUGHOOKINFO** structure has the following form:

```
typedef struct tagDEBUGHOOKINFO {
    HMODULE hModuleHook;
    LPARAM  reserved;
    LPARAM  lParam;
    WPARAM  wParam;
    int     code;
} DEBUGHOOKINFO;
```

Return Value The callback function should return TRUE to prevent the system from calling another filter. Otherwise, the callback function must pass the filter information to the **CallNextHookEx** function.

Comments An application must install this callback function by specifying the WH_DEBUG filter type and the procedure-instance address of the callback function in a call to the **SetWindowsHookEx** function.

CallWndProc is a placeholder for the library-defined function name. The actual name must be exported by including it in an **EXPORTS** statement in the library's module-definition file.

See Also **CallNextHookEx, SetWindowsHookEx**

DefDriverProc 3.1

Syntax LRESULT DefDriverProc(dwDriverIdentifier, hdrvr, uMsg, lParam1, lParam2)

function DefDriverProc(DriverIdentifier: Longint; DriverId: THandle; Message: Word; lParam1, lParam2: Longint): Longint;

The **DefDriverProc** function provides default processing for any messages not processed by an installable driver.

Parameters

dwDriverIdentifier	Identifies an installable driver. This parameter must have been obtained by a previous call to the **OpenDriver** function.
hdrvr	Identifies the installable driver.
uMsg	Specifies the message to be processed.
lParam1	Specifies 32 bits of additional message-dependent information.
lParam2	Specifies 32 bits of additional message-dependent information.

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

Comments The **DefDriverProc** function processes messages that are not handled by the **DriverProc** function.

See Also **OpenDriver, SendDriverMessage**

DirectedYield 3.1

Syntax void DirectedYield(htask)

procedure DirectedYield(Task: TTask);

The **DirectedYield** function puts the current task to sleep and awakens the given task.

Parameters *htask* Specifies the task to be executed.

Return Value This function does not return a value.

Comments When relinquishing control to other applications (that is, when exiting hard mode), a Windows-based debugger should call **DirectedYield,** identifying the handle of the task being debugged. This ensures that the debugged application runs next and that messages received during debugging are processed by the appropriate windows.

The Windows scheduler executes a task only when there is an event waiting for it, such as a paint message, or a message posted in the message queue.

If an application uses **DirectedYield** for a task with no events scheduled, the task will not be executed. Instead, Windows searches the task queue. In some cases, however, you may want the application to force a specific task to be scheduled. The application can do this by calling the **PostAppMessage** function, specifying a WM_NULL message identifier. Then, when the application calls **DirectedYield**, the scheduler will run the task regardless of the task's event status.

DirectedYield starts the task identified by *htask* at the location where it left off. Typically, debuggers should use **TaskSwitch** instead of **DirectedYield,** because **TaskSwitch** can start a task at any address.

DirectedYield returns when the current task is reawakened. This occurs when the task identified by *htask* waits for messages or uses the **Yield** or **DirectedYield** function. Execution will continue as before the task switch.

DirectedYield is located in KRNL286.EXE and KRNL386.EXE and is available in Windows versions 3.0 and 3.1.

See Also **PostAppMessage, TaskSwitch, TaskGetCSIP, TaskSetCSIP, Yield**

DlgDirSelectComboBoxEx 3.0

Syntax BOOL DlgDirSelectComboBoxEx(hwndDlg, lpszPath, cbPath, idComboBox)

function DlgDirSelectComboBoxEx(Dlg: HWnd; Path: PChar; cbPath: Integer; ComboBox: Integer): Bool;

The **DlgDirSelectComboBoxEx** function retrieves the current selection from the list box of a combo box. The list box should have been filled by the **DlgDirListComboBox** function, and the selection should be a drive letter, a file, or a directory name.

Parameters

hwndDlg	Identifies the dialog box that contains the combo box.
lpszPath	Points to a buffer that receives the selected path or filename.
cbPath	Specifies the length, in bytes, of the path or filename pointed to by the *lpszPath* parameter. This value should not be larger than 128.
idComboBox	Specifies the integer identifier of the combo box in the dialog box.

Return Value The return value is nonzero if the current combo box selection is a directory name. Otherwise, it is zero.

Comments The **DlgDirSelectComboBoxEx** function does not allow more than one filename to be returned from a combo box.

If the current selection is a directory name or drive letter, **DlgDirSelectComboBoxEx** removes the enclosing square brackets (and hyphens, for drive letters) so that the name or letter is ready to be inserted into a new path or filename. If there is no selection, the contents of buffer pointed to by the *lpszPath* parameter do not change.

DlgDirSelectComboBoxEx sends CB_GETCURSEL and CB_GETLBTEXT messages to the combo box.

See Also **DlgDirList, DlgDirListComboBox, DlgDirSelect, DlgDirSelectEx, DlgDirSelectComboBox**

DlgDirSelectEx 2.x

Syntax BOOL DlgDirSelectEx(hwndDlg, lpszPath, cbPath, idListBox)

function DlgDirSelectEx(Dlg: HWnd; Path: PChar; cbPath: Integer; ListBox: Integer): Bool;

The **DlgDirSelectEx** function retrieves the current selection from a list box. The specified list box should have been filled by the **DlgDirList** function, and the selection should be a drive letter, a file, or a directory name.

Parameters

hwndDlg	Identifies the dialog box that contains the list box.
lpszPath	Points to a buffer that receives the selected path or filename.
cbPath	Specifies the length, in bytes, of the path or filename pointed to by the *lpszPath* parameter. This value should not be larger than 128.
idListBox	Specifies the integer identifier of a list box in the dialog box.

Return Value The return value is nonzero if the current list box selection is a directory name. Otherwise, it is zero.

Comments If the current selection is a directory name or drive letter, **DlgDirSelectEx** removes the enclosing square brackets (and hyphens, for drive letters) so that the name or letter is ready to be inserted into a new path or filename. If there is no selection, the contents of buffer pointed to by the *lpszPath* parameter do not change.

The **DlgDirSelectEx** function does not allow more than one filename to be returned from a list box.

The list box must not be a multiple-selection list box. If it is, this function will not return a zero value and *lpszPath* will remain unchanged.

DlgDirSelectEx sends LB_GETCURSEL and LB_GETTEXT messages to the list box.

See Also **DlgDirList, DlgDirListComboBox, DlgDirSelect, DlgDirSelectComboBox**

DragAcceptFiles 3.1

Syntax
#include <shellapi.h>
void DragAcceptFiles(hwnd, fAccept)

procedure DragAcceptFiles(Wnd: HWnd; Accept: Bool);

The **DragAcceptFiles** function registers whether a given window accepts dropped files.

Parameters

hwnd Identifies the window registering whether it accepts dropped files.

fAccept Specifies whether the window specified by the *hwnd* parameter accepts dropped files. An application should set this value to TRUE to accept dropped files or FALSE to discontinue accepting dropped files.

Return Value This function does not return a value.

Comments When an application calls **DragAcceptFiles** with *fAccept* set to TRUE, Windows File Manager (WINFILE.EXE) sends the specified window a WM_DROPFILES message each time the user drops a file in that window.

DragFinish 3.1

Syntax
#include <shellapi.h>
void DragFinish(hDrop)

procedure DragFinish(Drop: THandle);

The **DragFinish** function releases memory that Windows allocated for use in transferring filenames to the application.

Parameters

hDrop Identifies the internal data structure that describes dropped files. This handle is passed to the application in the *wParam* parameter of the WM_DROPFILES message.

Return Value This function does not return a value.

DragQueryFile 3.1

Syntax #include <shellapi.h>
UINT DragQueryFile(hDrop, iFile, lpszFile, cb)

function DragQueryFile(Drop: THandle; FileIndex: Word; FileName: PChar; cb: Word): Word;

The **DragQueryFile** function retrieves the number of dropped files and their filenames.

Parameters

hDrop Identifies the internal data structure containing filenames for the dropped files. This handle is passed to the application in the *wParam* parameter of the WM_DROPFILES message.

iFile Specifies the index of the file to query. The index of the first file is 0. If the value of the *iFile* parameter is –1, **DragQueryFile** returns the number of files dropped. If the value of the *iFile* parameter is between zero and the total number of files dropped, **DragQueryFile** copies the filename corresponding to that value to the buffer pointed to by the *lpszFile* parameter.

lpszFile Points to a null-terminated string that contains the filename of a dropped file when the function returns. If this parameter is NULL and the *iFile* parameter specifies the index for the name of a dropped file, **DragQueryFile** returns the required size, in bytes, of the buffer for that filename.

cb Specifies the size, in bytes, of the *lpszFile* buffer.

Return Value When the function copies a filename to the *lpszFile* buffer, the return value is the number of bytes copied. If the *iFile* parameter is 0xFFFF, the return value is the number of dropped files. If *iFile* is between zero and the total number of dropped files and if *lpszFile* is NULL, the return value is the required size of the *lpszFile* buffer.

See Also **DragQueryPoint**

DragQueryPoint 3.1

Syntax #include <shellapi.h>
BOOL DragQueryPoint(hDrop, lppt)

function DragQueryPoint(Drop: THandle; var Pt: TPoint): Bool;

The **DragQueryPoint** function retrieves the window coordinates of the cursor when a file is dropped.

Parameters

hDrop Identifies the internal data structure that describes the dropped file. This structure is returned in the *wParam* parameter of the WM_DROPFILES message.

lppt Points to a **POINT** structure that the function fills with the coordinates of the position at which the cursor was located when the file was dropped. The **POINT** structure has the following form:

```
typedef struct tagPOINT {   /* pt */
   int x;
   int y;
} POINT;
```

Return Value The return value is nonzero if the file is dropped in the client area of the window. Otherwise, it is zero.

Comments The **DragQueryPoint** function fills the **POINT** structure with the coordinates of the position at which the cursor was located when the user released the left mouse button. The window for which coordinates are returned is the window that received the WM_DROPFILES message.

See Also **DragQueryFile**

DriverProc 3.1

Syntax LRESULT CALLBACK DriverProc(dwDriverIdentifier, hDriver, wMessage, lParam1, lParam2)

The **DriverProc** function processes the specified message.

Parameters

dwDriverIdentifier Specifies an identifier of the installable driver.

hDriver Identifies the installable driver. This parameter is a unique handle that Windows assigns to the driver.

wMessage Identifies a message that the driver must process. Following are the messages that Windows or an application can send to an installable driver:

Message	Description
DRV_CLOSE	Notifies the driver that it should decrement (decrease by one) its usage count and unload the driver if the count is zero.
DRV_CONFIGURE	Notifies the driver that it should display a custom-configuration dialog box. (This message should be sent only if the driver returns a nonzero value when the DRV_QUERYCONFIGURE message is processed.)
DRV_DISABLE	Notifies the driver that its allocated memory is about to be freed.
DRV_ENABLE	Notifies the driver that it has been loaded or reloaded, or that Windows has been enabled.
DRV_FREE	Notifies the driver that it will be discarded.
DRV_INSTALL	Notifies the driver that it has been successfully installed.
DRV_LOAD	Notifies the driver that it has been successfully loaded.
DRV_OPEN	Notifies the driver that it is about to be opened.
DRV_POWER	Notifies the driver that the device's power source is about to be turned off or turned on.
DRV_QUERYCONFIGURE	Determines whether the driver supports the DRV_CONFIGURE message. The message displays a private configuration dialog box.
DRV_REMOVE	Notifies the driver that it is about to be removed from the system.

lParam1 Specifies the first message parameter.

lParam2 Specifies the second message parameter.

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

Comments The **DriverProc** function is the main function within a Windows installable driver; it is supplied by the driver developer.

When the *wMessage* parameter is DRV_OPEN, *lParam1* is the string following the driver filename from the SYSTEM.INI file and *lParam2* is the value given as the *lParam* parameter in the call to the **OpenDriver** function.

When the *wMessage* parameter is DRV_CLOSE, *lParam1* and *lParam2* are the same values as the *lParam1* and *lParam2* parameters in the call to the **CloseDriver** function.

See Also **CloseDriver, OpenDriver**

EnableCommNotification 3.1

Syntax BOOL EnableCommNotification(idComDev, hwnd, cbWriteNotify, cbOutQueue)

function EnableCommNotification(idComDev: Integer; hwnd: HWnd; cbWriteNotify, cbOutQueue: Integer): Bool;

The **EnableCommNotification** function enables or disables WM_COMMNOTIFY message posting to the given window.

Parameters

idComDev	Specifies the communications device that is posting notification messages to the window identified by the *hwnd* parameter. The **OpenComm** function returns the value for the *idComDev* parameter.
hwnd	Identifies the window whose WM_COMMNOTIFY message posting will be enabled or disabled. If this parameter is NULL, **EnableCommNotification** disables message posting to the current window.
cbWriteNotify	Indicates the number of bytes the COM driver must write to the application's input queue before sending a notification message. The message signals the application to read information from the input queue.

cbOutQueue	Indicates the minimum number of bytes in the output queue. When the number of bytes in the output queue falls below this number, the COM driver sends the application a notification message, signaling it to write information to the output queue.

Return Value The return value is nonzero if the function is successful. Otherwise, it is zero, indicating an invalid COM port identifier, a port that is not open, or a function not supported by COMM.DRV.

Comments If an application specifies –1 for the *cbWriteNotify* parameter, the WM_COMMNOTIFY message is sent to the specified window for CN_EVENT and CN_TRANSMIT notifications but not for CN_RECEIVE notifications. If –1 is specified for the *cbOutQueue* parameter, CN_EVENT and CN_RECEIVE notifications are sent but CN_TRANSMIT notifications are not.

If a timeout occurs before as many bytes as specified by the *cbWriteNotify* parameter are written to the input queue, a WM_COMMNOTIFY message is sent with the CN_RECEIVE flag set. When this occurs, another message will not be sent until the number of bytes in the input queue falls below the number specified in the *cbWriteNotify* parameter. Similarly, a WM_COMMNOTIFY message in which the CN_RECEIVE flag is set is sent only when the output queue is larger than the number of bytes specified in the *cbOutQueue* parameter.

The Windows 3.0 version of COMM.DRV does not support this function.

EnableScrollBar 3.1

Syntax BOOL EnableScrollBar(hwnd, fnSBFlags, fuArrowFlags)

function EnableScrollBar(hwnd: HWnd; fnSBFlags: Integer; fuArrowFlags: Word): Bool;

The **EnableScrollBar** function enables or disables one or both arrows of a scroll bar.

Parameters

hwnd	Identifies a window or a scroll bar, depending on the value of the *fnSBFlags* parameter.
fnSBFlags	Specifies the scroll bar type. This parameter can be one of the following values:

Value	Meaning
SB_BOTH	Enables or disables the arrows of the horizontal and vertical scroll bars associated with the given window. The *hwnd* parameter identifies the window.
SB_CTL	Identifies the scroll bar as a scroll bar control. The *hwnd* parameter must identify a scroll bar control.
SB_HORZ	Enables or disables the arrows of the horizontal scroll bar associated with the given window. The *hwnd* parameter identifies the window.
SB_VERT	Enables or disables the arrows of the vertical scroll bar associated with the given window. The *hwnd* parameter identifies the window.

fuArrowFlags Specifies whether the scroll bar arrows are enabled or disabled, and which arrows are enabled or disabled. This parameter can be one of the following values:

Value	Meaning
ESB_ENABLE_BOTH	Enables both arrows of a scroll bar.
ESB_DISABLE_LTUP	Disables the left arrow of a horizontal scroll bar, or the up arrow of a vertical scroll bar.
ESB_DISABLE_RTDN	Disables the right arrow of a horizontal scroll bar, or the down arrow of a vertical scroll bar.
ESB_DISABLE_BOTH	Disables both arrows of a scroll bar.

Return Value The return value is nonzero if the arrows are enabled or disabled as specified. Otherwise, it is zero, indicating that the arrows are already in the requested state or that an error occurred.

Example The following example enables an edit control's vertical scroll bar when the control receives the input focus, and disables the scroll bar when the control loses the focus:

```
case EN_SETFOCUS:
    EnableScrollBar(hwndMLEdit, SB_VERT, ESB_ENABLE_BOTH);
    break;

case EN_KILLFOCUS:
    EnableScrollBar(hwndMLEdit, SB_VERT, ESB_DISABLE_BOTH);
    break;
```

See Also **ShowScrollBar**

EndDoc 3.1

Syntax int EndDoc(hdc)

function EndDoc(DC: HDC): Integer;

The **EndDoc** function ends a print job. This function replaces the ENDDOC printer escape for Windows version 3.1.

Parameters *hdc* Identifies the device context for the print job.

Return Value The return value is greater than or equal to zero if the function is successful. Otherwise, it is less than zero.

Comments An application should call the **EndDoc** function immediately after finishing a successful print job. To terminate a print job because of an error or if the user chooses to cancel the job, an application should call the **AbortDoc** function.

Do not use the **EndDoc** function inside metafiles.

See Also **AbortDoc, Escape, StartDoc**

EndPage 3.1

Syntax int EndPage(hdc)

function EndPage(DC: HDC): Integer;

The **EndPage** function signals the device that the application has finished writing to a page. This function is typically used to direct the driver to advance to a new page.

This function replaces the **NEWFRAME** printer escape for Windows 3.1. Unlike **NEWFRAME**, this function is always called after printing a page.

Parameters *hdc* Identifies the device context for the print job.

Return Value The return value is greater than or equal to zero if the function is successful. Otherwise, it is an error value.

Errors If the function fails, it returns one of the following error values:

Value	Meaning
SP_ERROR	General error.
SP_APPABORT	Job was terminated because the application's print-canceling function returned zero.
SP_USERABORT	User terminated the job by using Windows Print Manager (PRINTMAN.EXE).
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.

Comments The **ResetDC** function can be used to change the device mode, if necessary, after calling the **EndPage** function.

See Also **Escape, ResetDC, StartPage**

EnumFontFamilies 3.1

Syntax int EnumFontFamilies(hdc, lpszFamily, fntenmprc, lParam)

function EnumFontFamilies(DC: HDC; Family: PChar; EnumProc: TFontEnumProc; Data: PChar): Integer;

The **EnumFontFamilies** function enumerates the fonts in a specified font family that are available on a given device. **EnumFontFamilies** continues until there are no more fonts or the callback function returns zero.

Parameters

hdc Identifies the device context.

lpszFamily Points to a null-terminated string that specifies the family name of the desired fonts. If this parameter is NULL, the **EnumFontFamilies** function selects and enumerates one font from each available font family.

fntenmprc Specifies the procedure-instance address of the application-defined callback function. The address must be created by the **MakeProcInstance** function. For more information about the callback function, see the description of the **EnumFontFamProc** callback function.

lParam Specifies a 32-bit application-defined value that is passed to the callback function along with the font information.

Return Value The return value specifies the last value returned by the callback function, if the function is successful. This value depends on which font families are available for the given device.

Comments The **EnumFontFamilies** function differs from the **EnumFonts** function in that it retrieves the style names associated with a TrueType font. Using **EnumFontFamilies**, an application can retrieve information about unusual font styles (for example, Outline) that cannot be enumerated by using the **EnumFonts** function. Applications should use **EnumFontFamilies** instead of **EnumFonts**.

For each font having the font name specified by the *lpszFamily* parameter, the **EnumFontFamilies** function retrieves information about that font and passes it to the function pointed to by the *fntenmprc* parameter. The application-supplied callback function can process the font information, as necessary.

Example The following example uses the **MakeProcInstance** function to create a pointer to the callback function for the **EnumFontFamilies** function. The **FreeProcInstance** function is called when enumeration is complete. Because the second parameter is NULL, **EnumFontFamilies** enumerates one font from each family that is available in the given device context. The aFontCount variable points to an array that is used inside the callback function.

```
FONTENUMPROC lpEnumFamCallBack;
int aFontCount[] = { 0, 0, 0 };

lpEnumFamCallBack = (FONTENUMPROC) MakeProcInstance(
    (FARPROC) EnumFamCallBack, hAppInstance);
EnumFontFamilies(hdc, NULL, lpEnumFamCallBack, (LPARAM) aFontCount);
FreeProcInstance((FARPROC) lpEnumFamCallBack);
```

See Also **EnumFonts, EnumFontFamProc**

EnumFontFamProc 3.1

Syntax int CALLBACK EnumFontFamProc(lpnlf, lpntm, FontType, lParam)

TFontEnumProc = TFarProc;

The **EnumFontFamProc** function is an application-defined callback function that retrieves information about available fonts.

Parameters *lpnlf* Points to a NEWLOGFONT structure that contains information about the logical attributes of the font. This structure is locally-defined and is identical to the Windows **LOGFONT** structure except for two new members. The NEWLOGFONT structure has the following form:

```
struct tagNEWLOGFONT {                  /* nlf */
    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];
    BYTE  lfFullName[2 * LF_FACESIZE]; /* TrueType only
*/
    BYTE  lfStyle[LF_FACESIZE];        /* TrueType only
*/
}NEWLOGFONT;
```

The **lfFullName** and **lfStyle** members are appended to a **LOGFONT** structure when a TrueType font is enumerated in the **EnumFontFamProc** function.

The **lfFullName** member is a character array specifying the full name for the font. This name contains the font name and style name.

The **lfStyle** member is a character array specifying the style name for the font.

For example, when bold italic Arial®is enumerated, the last three members of the NEWLOGFONT structure contain the following strings:

lfFaceName = "Arial";
lfFullName = "Arial Bold Italic";
lfStyle = "Bold Italic";

lpntm Points to a **NEWTEXTMETRIC** structure that contains information about the physical attributes of the font, if the font is a TrueType font. If the font is not a TrueType font, this parameter points to a **TEXTMETRIC** structure.

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

```
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 **TEXTMETRIC** structure is identical to **NEWTEXTMETRIC** except that it does not include the last four members.

FontType Specifies the type of the font. This parameter can be a combination of the following masks:

DEVICE_FONTTYPE
RASTER_FONTTYPE
TRUETYPE_FONTTYPE

lParam Points to the application-defined data passed by **EnumFontFamilies**.

Return Value This function must return a nonzero value to continue enumeration; to stop enumeration, it must return zero.

Comments An application must register this callback function by passing its address to the **EnumFontFamilies** function. The **EnumFontFamProc** function is a placeholder for the application-defined function name. The actual name must be exported by including it in an **EXPORTS** statement in the application's module-definition (.DEF) file.

The AND (&) operator can be used with the RASTER_FONTTYPE, DEVICE_FONTTYPE, and TRUETYPE_FONTTYPE constants to determine the font type. If the RASTER_FONTTYPE bit is set, the font is a raster font. If the TRUETYPE_FONTTYPE bit is set, the font is a TrueType font. If neither bit is set, the font is a vector font. A third mask, DEVICE_FONT-TYPE, is set when a device (for example, a laser printer) supports downloading TrueType fonts; it is zero if the font is not a device font. (Any device can support device fonts, including display adapters and dot-matrix printers.) An application can also use the DEVICE_FONT-TYPE mask to distinguish GDI-supplied raster fonts from device-supplied fonts. GDI can simulate bold, italic, underline, and strikeout attributes for GDI-supplied raster fonts, but not for device-supplied fonts.

See Also **EnumFontFamilies, EnumFonts**

EnumFontsProc 3.1

Syntax int CALLBACK EnumFontsProc(lplf, lpntm, FontType, lpData)

TOldFontEnumProc = TFarProc;

The **EnumFontsProc** function is an application-defined callback function that processes font data from the **EnumFonts** function.

Parameters *lplf* Points to a **LOGFONT** structure that contains information about the logical attributes of the font. The **LOGFONT** structure has the following form:

```
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;
```

lpntm Points to a **NEWTEXTMETRIC** structure that contains information about the physical attributes of the font, if the font is a TrueType font. If the font is not a TrueType font, this parameter points to a **TEXTMETRIC** structure.

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

```
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 **TEXTMETRIC** structure is identical to **NEWTEXTMETRIC** except that it does not include the last four members.

FontType Specifies the type of the font. This parameter can be a combination of the following masks:

DEVICE_FONTTYPE
RASTER_FONTTYPE
TRUETYPE_FONTTYPE

lpData Points to the application-defined data passed by the **EnumFonts** function.

Return Value This function must return a nonzero value to continue enumeration; to stop enumeration, it must return zero.

Comments An application must register this callback function by passing its address to the **EnumFonts** function. The **EnumFontsProc** function is a placeholder for the application-defined function name. The actual name must be exported by including it in an **EXPORTS** statement in the application's module-definition (.DEF) file.

The AND (&) operator can be used with the RASTER_FONTTYPE, DEVICE_FONTTYPE, and TRUETYPE_FONTTYPE constants to determine the font type. If the RASTER_FONTTYPE bit is set, the font is a raster font. If the TRUETYPE_FONTTYPE bit is set, the font is a TrueType font. If neither bit is set, the font is a vector font. A third mask, DEVICE_FONTTYPE, is set when a device (for example, a laser printer) supports downloading TrueType fonts; it is zero if the device is a display adapter, dot-matrix printer, or other raster device. An application can also use the DEVICE_FONTTYPE mask to distinguish GDI-supplied raster fonts from device-supplied fonts. GDI can simulate bold, italic, underline, and strikeout attributes for GDI-supplied raster fonts, but not for device-supplied fonts.

See Also **EnumFonts, EnumFontFamilies**

EnumMetaFileProc 3.1

Syntax int CALLBACK EnumMetaFileProc(hdc, lpht, lpmr, cObj, lParam)

The **EnumMetaFileProc** function is an application-defined callback function that processes metafile data from the **EnumMetaFile** function.

Parameters

hdc	Identifies the special device context that contains the metafile.
lpht	Points to a table of handles associated with the objects (pens, brushes, and so on) in the metafile.
lpmr	Points to a metafile record contained in the metafile.
cObj	Specifies the number of objects with associated handles in the handle table.
lParam	Points to the application-defined data.

Return Value The callback function must return a nonzero value to continue enumeration; to stop enumeration, it must return zero.

Comments An application must register this callback function by passing its address to the **EnumMetaFile** function.

The **EnumMetaFileProc** function is a placeholder for the application-defined function name. The actual name must be exported by including it in an **EXPORTS** statement in the application's module-definition (.DEF) file.

See Also **EnumMetaFile**

EnumObjectsProc 3.1

Syntax int CALLBACK EnumObjectsProc(lpLogObject, lpData)

The **EnumObjectsProc** function is an application-defined callback function that processes object data from the **EnumObjects** function.

Parameters

lpLogObject Points to a **LOGPEN** or **LOGBRUSH** structure that contains information about the attributes of the object.

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

```
typedef struct tagLOGPEN {  /* lgpn */
    UINT     lopnStyle;
    POINT    lopnWidth;
    COLORREF lopnColor;
} LOGPEN;
```

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

```
typedef struct tagLOGBRUSH {    /* lb */
    UINT     lbStyle;
    COLORREF lbColor;
    int      lbHatch;
} LOGBRUSH;
```

lpData Points to the application-defined data passed by the **EnumObjects** function.

Return Value This function must return a nonzero value to continue enumeration; to stop enumeration, it must return zero.

Comments An application must register this callback function by passing its address to the **EnumObjects** function. The **EnumObjectsProc** function is a placeholder for the application-supplied function name. The actual name must be exported by including it in an **EXPORTS** statement in the application's module-definition (.DEF) file.

Example The following example retrieves the number of horizontally hatched brushes and fills **LOGBRUSH** structures with information about each of them:

```
#define MAXBRUSHES 50

GOBJENUMPROC lpProcCallback;
HGLOBAL hglbl;
LPBYTE lpbCountBrush;

lpProcCallback = (GOBJENUMPROC) MakeProcInstance(
    (FARPROC) Callback, hinst);

hglbl = GlobalAlloc(GMEM_FIXED, sizeof(LOGBRUSH)
    * MAXBRUSHES);
lpbCountBrush = (LPBYTE) GlobalLock(hglbl);
*lpbCountBrush = 0;
EnumObjects(hdc, OBJ_BRUSH, lpProcCallback,
    (LPARAM) lpbCountBrush);

FreeProcInstance((FARPROC) lpProcCallback);

intFARPASCALCallback(LPLOGBRUSHlpLogBrush,LPBYTEpbData)
{
    /*
     * The pbData parameter contains the number of horizontally
     * hatched brushes; the lpDest parameter is set to follow the
     * byte reserved for pbData and the LOGBRUSH structures that
     * have been filled with brush information.
     */

    LPLOGBRUSH lpDest =
        (LPLOGBRUSH) (pbData + 1 + (*pbData * sizeof(LOGBRUSH)));

    if (lpLogBrush->lbStyle ==
            BS_HATCHED && /* if horiz hatch */
            lpLogBrush->lbHatch == HS_HORIZONTAL) {
        *lpDest++ = *lpLogBrush; /* fills structure with brush info */
        (*pbData) ++;            /* increments brush count          */
        if (*pbData >= MAXBRUSHES)
            return 0;
    }

    return 1;
}
```

See Also **EnumObjects, FreeProcInstance, GlobalAlloc, GlobalLock, MakeProcInstance**

EnumPropFixedProc 2.x

Syntax BOOL CALLBACK EnumPropFixedProc(hwnd, lpsz, hData)

The **EnumPropFixedProc** function is an application-defined callback function that receives a window's property data as a result of a call to the **EnumProps** function.

Parameters

hwnd	Identifies the handle of the window that contains the property list.
lpsz	Points to the null-terminated string associated with the property data identified by the *hData* parameter. The application specified the string and data in a previous call to the **SetProp** function. If the application passed an atom instead of a string to **SetProp**, the *lpsz* parameter contains the atom in the low-order word and zero in the high-order word.
hData	Identifies the property data.

Return Value The callback function must return TRUE to continue enumeration; it must return FALSE to stop enumeration.

Comments This form of the property-enumeration callback function should be used in applications and dynamic-link libraries with fixed data segments and in dynamic libraries with movable data segments that do not contain a stack.

The following restrictions apply to the callback function:

- The callback function must not yield control or do anything that might yield control to other tasks.
- The callback function can call the **RemoveProp** function. However, **RemoveProp** can remove only the property passed to the callback function through the callback function's parameters.
- The callback function should not attempt to add properties.

The **EnumPropFixedProc** function is a placeholder for the application-defined function name. The actual name must be exported by including it in an **EXPORTS** statement in the application's module-definition (.DEF) file.

See Also **EnumPropMovableProc, EnumProps, RemoveProp, SetProp**

EnumPropMovableProc 2.x

Syntax BOOL CALLBACK EnumPropMovableProc(hwnd, lpsz, hData)

The **EnumPropMovableProc** function is an application-defined callback function that receives a window's property data as a result of a call to the **EnumProps** function.

Parameters

hwnd	Identifies the handle of the window that contains the property list.
lpsz	Points to the null-terminated string associated with the data identified by the *hData* parameter. The application specified the string and data in a previous call to the **SetProp** function. If the application passed an atom instead of a string to **SetProp**, the *lpsz* parameter contains the atom.
hData	Identifies the property data.

Return Value The callback function must return TRUE to continue enumeration; to stop enumeration, it must return FALSE.

Comments This form of the property-enumeration callback function should be used in applications with movable data segments and in dynamic libraries whose movable data segments also contain a stack. This form is required since movement of the data will invalidate any long pointer to a variable on the stack, such as the *lpsz* parameter. The data segment typically moves if the callback function allocates more space in the local heap than is currently available.

The following restrictions apply to the callback function:

- The callback function must not yield control or do anything that might yield control to other tasks.
- The callback function can call the **RemoveProp** function. However, **RemoveProp** can remove only the property passed to the callback function through the callback function's parameters.
- The callback function should not attempt to add properties.

The **EnumPropMovableProc** function is a placeholder for the application-defined function name. The actual name must be exported by including it in an EXPORTS statement in the application's module-definition (.DEF) file.

See Also **EnumPropFixedProc, EnumProps, RemoveProp, SetProp**

EnumTaskWndProc 2.x

Syntax BOOL CALLBACK EnumTaskWndProc(hwnd, lParam)

The **EnumTaskWndProc** function is an application-defined callback function that receives the window handles associated with a task as a result of a call to the **EnumTaskWindows** function.

Parameters

hwnd	Identifies a window associated with the task specified in the **EnumTaskWindows** function.
lParam	Specifies the application-defined value specified in the **EnumTaskWindows** function.

Return Value The callback function must return TRUE to continue enumeration; to stop enumeration, it must return FALSE.

Comments The callback function can carry out any desired task.

The **EnumTaskWndProc** function is a placeholder for the application-defined function name. The actual name must be exported by including it in an **EXPORTS** statement in the application's module-definition (.DEF) file.

See Also **EnumTaskWindows**

EnumWindowsProc 2.x

Syntax BOOL CALLBACK EnumWindowsProc(hwnd, lParam)

The **EnumWindowsProc** function is an application-defined callback function that receives parent window handles as a result of a call to the **EnumWindows** function.

Parameters

hwnd	Identifies a parent window.
lParam	Specifies the application-defined value specified in the **EnumWindows** function.

Return Value The callback function must return nonzero to continue enumeration; to stop enumeration, it must return zero.

Comments The callback function can carry out any desired task.

The **EnumWindowsProc** function is a placeholder for the application-defined function name. The actual name must be exported by including it in an **EXPORTS** statement in the application's module-definition (.DEF) file.

See Also **EnumWindows**

ExitWindowsExec 3.0

Syntax BOOL ExitWindowsExec(lpszExe, lpszParams)
function ExitWindowExec(Exe: PChar; Params: PChar): Bool;

The **ExitWindowsExec** function terminates Windows, runs a specified MS-DOS application, and then restarts Windows.

Parameters

lpszExe	Points to a null-terminated string specifying the path and filename of the executable file for the system to run after Windows has been terminated. This string must not be longer than 128 bytes (including the null terminating character).
lpszParams	Points to a null-terminated string specifying any parameters for the executable file specified by the *lpszExe* parameter. This string must not be longer than 127 bytes (including the null terminating character). This value can be NULL.

Return Value The return value is FALSE if the function fails. (The function could fail because of a memory-allocation error or if one of the applications in the system does not terminate.)

Comments The **ExitWindowsExec** function is typically used by installation programs to replace components of Windows which are active when Windows is running.

See Also **ExitWindows**

ExtractIcon 3.1

Syntax #include <shellapi.h>
HICON ExtractIcon(hinst, lpszExeName, iIcon)

function ExtractIcon(Inst: THandle; ExeFileName: PChar; IconIndex: Word): HIcon;

The **ExtractIcon** function retrieves the handle of an icon from a specified executable file, dynamic-link library (DLL), or icon file.

Parameters

hinst	Identifies the instance of the application calling the function.
lpszExeName	Points to a null-terminated string specifying the name of an executable file, dynamic-link library, or icon file.
iIcon	Specifies the index of the icon to be retrieved. If this parameter is zero, the function returns the handle of the first icon in the specified file. If the parameter is –1, the function returns the total number of icons in the specified file.

Return Value The return value is the handle of an icon if the function is successful. It is 1 if the file specified in the *lpszExeName* parameter is not an executable file, dynamic-link library, or icon file. Otherwise, it is NULL, indicating that the file contains no icons.

FindExecutable 3.1

Syntax #include <shellapi.h>
HINSTANCE FindExecutable(lpszFile, lpszDir, lpszResult)

function FindExecutable(FileName, Directory, Result: PChar): THandle;

The **FindExecutable** function finds and retrieves the executable filename that is associated with a specified filename.

Parameters

lpszFile	Points to a null-terminated string specifying a filename. This can be a document or executable file.
lpszDir	Points to a null-terminated string specifying the drive letter and path for the default directory.
lpszResult	Points to a buffer that receives the name of an executable file when the function returns. This null-terminated string

specifies the application that is started when the Open command is chosen from the File menu in File Manager.

Return Value The return value is greater than 32 if the function is successful. If the return value is less than or equal to 32, it specifies an error code.

Errors The **FindExecutable** function returns 31 if there is no association for the specified file type. The other possible error values are as follows:

Value	Meaning
0	System was out of memory, executable file was corrupt, or relocations were invalid.
2	File was not found.
3	Path was not found.
5	Attempt was made to dynamically link to a task, or there was a sharing or network-protection error.
6	Library required separate data segments for each task.
8	There was insufficient memory to start the application.
10	Windows version was incorrect.
11	Executable file was invalid. Either it was not a Windows application or there was an error in the .EXE image.
12	Application was designed for a different operating system.
13	Application was designed for MS-DOS 4.0.
14	Type of executable file was unknown.
15	Attempt was made to load a real-mode application (developed for an earlier version of Windows).
16	Attempt was made to load a second instance of an executable file containing multiple data segments that were not marked read-only.
19	Attempt was made to load a compressed executable file. The file must be decompressed before it can be loaded.
20	Dynamic-link library (DLL) file was invalid. One of the DLLs required to run this application was corrupt.
21	Application requires Microsoft Windows 32-bit extensions.

Comments The filename specified in the *lpszFile* parameter is associated with an executable file when an association has been registered between that file's filename extension and an executable file in the registration database. An application that produces files with a given filename extension typically associates the extension with an executable file when the application is installed.

See Also **RegQueryValue, ShellExecute**

FindText 3.1

Syntax #include <commdlg.h>
HWND FindText(lpfr)

function FindText(var FindReplace: TFindReplace): HWnd;

The **FindText** function creates a system-defined modeless dialog box that makes it possible for the user to find text within a document. The application must perform the search operation.

Parameters

lpfr Points to a **FINDREPLACE** structure that contains information used to initialize the dialog box. When the user makes a selection in the dialog box, the system fills this structure with information about the user's selection and then sends a message to the application. This message contains a pointer to the **FINDREPLACE** structure.

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

```
#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;
```

Return Value The return value is the window handle of the dialog box if the function is successful. Otherwise, it is NULL. An application can use this window handle to communicate with or to close the dialog box.

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

CDERR_FINDRESFAILURE
CDERR_INITIALIZATION
CDERR_LOCKRESFAILURE
CDERR_LOADRESFAILURE
CDERR_LOADSTRFAILURE

CDERR_MEMALLOCFAILURE
CDERR_MEMLOCKFAILURE
CDERR_NOHINSTANCE
CDERR_NOHOOK
CDERR_NOTEMPLATE
CDERR_STRUCTSIZE
FRERR_BUFFERLENGTHZERO

Comments The dialog box procedure for the Find dialog box passes user requests to the application through special messages. The *lParam* parameter of each of these messages contains a pointer to a **FINDREPLACE** structure. The procedure sends the messages to the window identified by the **hwndOwner** member of the **FINDREPLACE** structure. An application can register the identifier for these messages by specifying the "commdlg_FindReplace" string in a call to the **RegisterWindowMessage** function.

For the TAB key to function correctly, any application that calls the **FindText** function must also call the **IsDialogMessage** function in its main message loop. (The **IsDialogMessage** function returns a value that indicates whether messages are intended for the Find dialog box.)

If the hook function (to which the **lpfnHook** member of the **FINDREPLACE** structure points) processes the WM_CTLCOLOR message, this function must return a handle of the brush that should be used to paint the control background.

Example The following example initializes a **FINDREPLACE** structure and calls the **FindText** function to display the Find dialog box:

```
FINDREPLACE fr;

/* Set all structure fields to zero. */

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

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

hDlg = FindText(&fr);

break;
```

In addition to initializing the members of the **FINDREPLACE** structure and calling the **FindText** function, an application must register the special FINDMSGSTRING message and process messages from the dialog box.

The following example registers the message by using the **RegisterWindowMessage** function:

```
UINT uFindReplaceMsg;

/* Register the FindReplace message. */

uFindReplaceMsg = RegisterWindowMessage(FINDMSGSTRING);
```

After the application registers the FINDMSGSTRING message, it can process messages by using the **RegisterWindowMessage** return value. The following example processes messages for the Find dialog box and then calls its own SearchFile function to locate the string of text:

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

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

See Also **IsDialogMessage, RegisterWindowMessage, ReplaceText**

FMExtensionProc 3.1

Syntax #include <wfext.h>
HMENU FAR PASCAL FMExtensionProc(hwnd, wMsg, lParam)

TFM_Ext_Proc = function(Handle: HWnd; w: Word; 1: Longint): Longint;

The **FMExtensionProc** function, an application-defined callback function, processes menu commands and messages sent to a File Manager extension dynamic-link library (DLL).

Parameters

hwnd Identifies the File Manager window. An extension DLL should use this handle to specify the parent for any dialog boxes or message boxes that the DLL may display and to send request messages to File Manager.

wMsg Specifies the message. This parameter may be one of the following values:

Value	Meaning
1–99	Identifier for the menu item that the user selected.
FMEVENT_INITMENU	User selected the extension's menu.
FMEVENT_LOAD	File Manager is loading the extension DLL.
FMEVENT_SELCHANGE	Selection in File Manager's directory window, or Search Results window, changed.
FMEVENT_UNLOAD	File Manager is unloading the extension DLL.
FMEVENT_USER_REFRESH	User chose the Refresh command from the Window menu.

lParam Specifies 32 bits of additional message-dependent information.

Return Value The callback function should return the result of the message processing. The actual return value depends on the message that is processed.

Comments Whenever File Manager calls the **FMExtensionProc** function, it waits to refresh its directory windows (for changes in the file system) until after the function returns. This allows the extension to perform large numbers of file operations without excessive repainting by the File Manager. The extension does not need to send the FM_REFRESH_WINDOWS message to notify File Manager to repaint its windows.

FreeAllGDIMem 3.1

Syntax

```
#include <stress.h>
void FreeAllGDIMem(void)
```

```
procedure FreeAllGDIMem;
```

The **FreeAllGDIMem** function frees all memory allocated by the **AllocGDIMem** function.

Parameters This function has no parameters.

Return Value This function does not return a value.

See Also **AllocGDIMem**

FreeAllMem 3.1

Syntax #include <stress.h>
void FreeAllMem(void)

procedure FreeAllMem;

The **FreeAllMem** function frees all memory allocated by the **AllocMem** function.

Parameters This function has no parameters.

Return Value This function does not return a value.

See Also **AllocMem**

FreeAllUserMem 3.1

Syntax #include <stress.h>
void FreeAllUserMem(void)

procedure FreeAllUserMem;

The **FreeAllUserMem** function frees all memory allocated by the **AllocUserMem** function.

Parameters This function has no parameters.

Return Value This function does not return a value.

See Also **AllocUserMem**

GetAspectRatioFilterEx 3.1

Syntax BOOL GetAspectRatioFilterEx(hdc, lpAspectRatio)

function GetAspectRatioFilterEx(DC: HDC; Size: PSize): Bool;

The **GetAspectRatioFilterEx** function retrieves the setting for the current aspect-ratio filter. The aspect ratio is the ratio formed by a device's pixel width and height. Information about a device's aspect ratio is used in the creation, selection, and displaying of fonts. Windows provides a special filter, the aspect-ratio filter, to select fonts designed for a particular aspect ratio from all of the available fonts. The filter uses the aspect ratio specified by the **SetMapperFlags** function.

Parameters

hDC	Identifies the device context that contains the specified aspect ratio.
lpAspectRatio	Pointer to a **SIZE** structure where the current aspect ratio filter will be returned.

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

See Also **SetMapperFlags**

GetBitmapDimensionEx 2.x

Syntax BOOL GetBitmapDimensionEx(hBitmap, lpDimension)

function GetBitmapDimensionEx(BM: HBitmap; Size: PSize): Bool;

The **GetBitmapDimensionEx** function returns the dimensions of the bitmap previously set by the **SetBitmapDimensionEx** function. If no dimensions have been set, a default of 0,0 will be returned.

Parameters

hBitmap Identifies the bitmap.

lpDimension Points to a **SIZE** structure to which the dimensions are returned. The **SIZE** structure has the following form:

```
typedef struct tagSIZE {
    int cx;
    int cy;
} SIZE;
```

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

See Also **SetBitmapDimensionEx**

GetBoundsRect 3.1

Syntax UINT GetBoundsRect(hdc, lprcBounds, flags)

function GetBoundsRect(DC: HDC; var Bounds: TRect; Flags: Word): Word;

The **GetBoundsRect** function returns the current accumulated bounding rectangle for the specified device context.

Windows maintains two accumulated bounding rectangles—one for the application and one reserved for use by Windows. An application can query and set its own rectangle, but can only query the Windows rectangle.

Parameters

hdc — Identifies the device context to return the bounding rectangle for.

lprcBounds — Points to a buffer that will receive the current bounding rectangle. The application's rectangle is returned in logical coordinates, and the Windows rectangle is returned in screen coordinates.

flags — Specifies the type of information to return. This parameter can be either or both of the following values:

Value	Meaning
DCB_RESET	Forces the bounding rectangle to be cleared after it is returned.
DCB_WINDOWMGR	Queries the Windows bounding rectangle instead of the application's.

Return Value

The return value specifies the current state of the bounding rectangle if the function is successful. It can be a combination of the following values:

Value	Meaning
DCB_ACCUMULATE	Bounding rectangle accumulation is occurring.
DCB_RESET	Bounding rectangle is empty.
DCB_SET	Bounding rectangle is not empty.
DCB_ENABLE	Bounding accumulation is on.
DCB_DISABLE	Bounding accumulation is off.

Comments

To ensure that the bounding rectangle is empty, check both the DCB_RESET bit and the DCB_ACCUMULATE bit in the return value. If DCB_RESET is set and DCB_ACCUMULATE is not, the bounding rectangle is empty.

See Also **SetBoundsRect**

GetBrushOrgEx 3.1

Syntax BOOL GetBrushOrgEx(hDC, lpPoint)

function GetBrushOrgEx(DC: HDC; Point: PPoint): Bool;

The **GetBrushOrgEx** function retrieves the current brush origin for the given device context.

Parameters

hDC	Identifies the device context.
lpPoint	Points to a **POINT** structure to which the device coordinates of the brush origin are to be returned. The **POINT** structure has the following form:

```
typedef struct tagPOINT {    /* pt */
   int x;
   int y;
} POINT;
```

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

Comments The initial brush origin is at the coordinate (0,0).

See Also **SetBrushOrg**

GetCharABCWidths 3.1

Syntax BOOL GetCharABCWidths(hdc, uFirstChar, uLastChar, lpabc)

function GetCharABCWidths(hdc: HDC; uFirstChar, uLastChar: Word; var lpabc: TABC): Bool;

The **GetCharABCWidths** function retrieves the widths of consecutive characters in a specified range from the current TrueType font. The widths are returned in logical units. This function succeeds only with TrueType fonts.

Parameters

hdc	Identifies the device context.
uFirstChar	Specifies the first character in the range of characters from the current font for which character widths are returned.
uLastChar	Specifies the last character in the range of characters from the current font for which character widths are returned.

lpabc	Points to an array of **ABC** structures that receive the character widths when the function returns. This array must contain at least as many **ABC** structures as there are characters in the range specified by the *uFirstChar* and *uLastChar* parameters.

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

Comments The TrueType rasterizer provides ABC character spacing after a specific point size has been selected. "A" spacing is the distance that is added to the current position before placing the glyph. "B" spacing is the width of the black part of the glyph. "C" spacing is added to the current position to account for the white space to the right of the glyph. The total advanced width is given by A + B + C.

When the **GetCharABCWidths** function retrieves negative "A" or "C" widths for a character, that character includes underhangs or overhangs.

To convert the ABC widths to font design units, an application should create a font whose height (as specified in the **lfHeight** member of the **LOGFONT** structure) is equal to the value stored in the **ntmSizeEM** member of the **NEWTEXTMETRIC** structure. (The value of the **ntmSizeEM** member can be retrieved by calling the **EnumFontFamilies** function.)

The ABC widths of the default character are used for characters that are outside the range of the currently selected font.

To retrieve the widths of characters in non-TrueType fonts, applications should use the **GetCharWidth** function.

See Also **EnumFontFamilies, GetCharWidth**

GetClipCursor 3.1

Syntax void GetClipCursor(lprc)

procedure GetClipCursor(var Rect: TRect);

The **GetClipCursor** function retrieves the screen coordinates of the rectangle to which the cursor has been confined by a previous call to the **ClipCursor** function.

Parameters

lprc	Points to a **RECT** structure that receives the screen coordinates of the confining rectangle. The structure

receives the dimensions of the screen if the cursor is not confined to a rectangle. The **RECT** structure has the following form:

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

Return Value This function does not return a value.

See Also **ClipCursor, GetCursorPos**

GetCurrentPositionEx 3.1

Syntax BOOL GetCurrentPositionEx(hdc; lpPoint)

function GetCurrentPositionEx(DC: HDC; Point: PPoint): Bool;

The **GetCurrentPositionEx** function retrieves the current position in logical coordinates.

Parameters

hdc	Identifies the device context to get the current position from.
lpPoint	Points to a **POINT** structure that gets filled with the current position.

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

GetCursor 3.1

Syntax HCURSOR GetCursor(void)

function GetCursor: HCursor;

The **GetCursor** function retrieves the handle of the current cursor.

Parameters This function has no parameters.

Return Value The return value is the handle of the current cursor if the function is successful. Otherwise, it is NULL.

See Also **SetCursor**

GetDCEx 3.1

Syntax HDC GetDCEx(hwnd, hrgnClip, fdwOptions)

function GetDCEx(Wnd: HWnd; Clip: HRgn; Flags: Longint): HDC;

The **GetDCEx** function retrieves the handle of a device context for the given window. The device context can be used in subsequent graphics device interface (GDI) functions to draw in the client area.

This function, which is an extension to the **GetDC** function, gives an application more control over how and whether a device context for a window is clipped.

Parameters

hwnd Identifies the window where drawing will occur.

hrgnClip Identifies a clipping region that may be combined with the visible region of the client window.

fdwOptions Specifies how the device context is created. This parameter can be a combination of the following values:

Value	Meaning
DCX_CACHE	Returns a device context from the cache, rather than the OWNDC or CLASSDC window. Essentially overrides CS_OWNDC and CS_CLASSDC.
DCX_CLIPCHILDREN	Excludes the visible regions of all child windows below the window identified by the *hwnd* parameter.
DCX_CLIPSIBLINGS	Excludes the visible regions of all sibling windows above the window identified by the *hwnd* parameter.
DCX_EXCLUDERGN	Excludes the clipping region identified by the *hrgnClip* parameter from the visible region of the returned device context.
DCX_INTERSECTRGN	Intersects the clipping region identified by the *hrgnClip* parameter with the visible region of the returned device context.
DCX_LOCKWINDOWUPDATE	Allows drawing even if there is a **LockWindowUpdate** call in effect that would otherwise exclude this window. This value is used for drawing during tracking.

Value	Meaning
DCX_PARENTCLIP	Uses the visible region of the parent window, ignoring the parent window's WS_CLIPCHILDREN and WS_PARENTDC style bits. This value sets the device context's origin to the upper-left corner of the window identified by the *hwnd* parameter.
DCX_WINDOW	Returns a device context corresponding to the window rectangle rather than the client rectangle.

Return Value The return value is a handle of the device context for the specified window, if the function is successful. Otherwise, it is NULL.

Comments Unless the device context belongs to a window class, the **ReleaseDC** function must be called to release the context after drawing. Since only five common device contexts are available at any given time, failure to release a device context can prevent other applications from accessing a device context.

A device context belonging to the window's class is returned by the **GetDCEx** function if the CS_CLASSDC, CS_OWNDC, or CS_PARENTDC class style was specified in the **WNDCLASS** structure when the class was registered.

In order to obtain a cached device context, an application must specify DCX_CACHE. If DCX_CACHE is not specified and the window is neither CS_OWNDC nor CS_CLASSDC, this function returns NULL.

See Also **BeginPaint, GetDC, GetWindowDC, ReleaseDC**

GetDriverInfo 3.1

Syntax BOOL GetDriverInfo(hdrvr, lpdis)

function GetDriverInfo(hDriver: THandle; lpdis: PDriverInfoStruct): Bool;

The **GetDriverInfo** function retrieves information about an installable driver.

Parameters

hdrvr	Identifies the installable driver. This handle must be retrieved by the **OpenDriver** function.

lpdis Points to a **DRIVERINFOSTRUCT** structure that receives the driver information. The **DRIVERINFOSTRUCT** structure has the following form:

```
typedef struct tagDRIVERINFOSTRUCT {     /* drvinfst */
    UINT      length;
    HDRVR     hDriver;
    HINSTANCE hModule;
    char      szAliasName[128];
} DRIVERINFOSTRUCT;
```

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

GetDriverModuleHandle 3.1

Syntax HINSTANCE GetDriverModuleHandle(hdrvr)

function GetDriverModuleHandle(Driver: THandle): THandle;

The **GetDriverModuleHandle** function retrieves the instance handle of a module that contains an installable driver.

Parameters *hdrvr* Identifies the installable driver. This parameter must be retrieved by the **OpenDriver** function.

Return Value The return value is an instance handle of the driver module if the function is successful. Otherwise, it is NULL.

See Also **OpenDriver**

GetExpandedName 3.1

Syntax #include <lzexpand.h>
int GetExpandedName(lpszSource, lpszBuffer)

function GetExpandedName(Source, Buffer: PChar): Integer;

The **GetExpandedName** function retrieves the original name of a compressed file if the file was compressed with the COMPRESS.EXE utility and the **/r** option was specified.

Parameters

lpszSource	Points to a string that specifies the name of a compressed file.
lpszBuffer	Points to a buffer that receives the name of the compressed file.

Return Value The return value is TRUE if the function is successful. Otherwise, it is an error value that is less than zero, and it may be LZERROR_BADINHANDLE, which means that the handle identifying the source file was not valid.

Example The following example uses the **GetExpandedName** function to retrieve the original filename of a compressed file:

```
char szSrc[] = {"readme.cmp"};
char szFileName[128];
OFSTRUCT ofStrSrc;
OFSTRUCT ofStrDest;
HFILE hfSrcFile, hfDstFile, hfCompFile;
int cbRead;
BYTE abBuf[512];

/* Open the compressed source file. */

hfSrcFile = OpenFile(szSrc, &ofStrSrc, OF_READ);

/*
 * Initialize internal data structures for the decompression
 * operation.
 */

hfCompFile = LZInit(hfSrcFile);

/* Retrieve the original name for the compressed file. */

GetExpandedName(szSrc, szFileName);

/* Create the destination file using the original name. */

hfDstFile = LZOpenFile(szFileName, &ofStrDest, OF_CREATE);
```

```
/* Copy the compressed source file to the destination file. */

do {
    if ((cbRead = LZRead(hfCompFile, abBuf, sizeof(abBuf))) > 0)
        _lwrite(hfDstFile, abBuf, cbRead);
    else {
        .
        . /* handle error condition */
        .
    }
} while (cbRead == sizeof(abBuf));

/* Close the files. */

LZClose(hfSrcFile);
LZClose(hfDstFile);
```

Comments This function retrieves the original filename from the header of the compressed file. If the source file is not compressed, the filename to which *lpszSource* points is copied to the buffer to which *lpszBuffer* points.

If the /**r** option was not set when the file was compressed, the string in the buffer to which *lpszBuffer* points is invalid.

GetFileResource 3.1

Syntax #include <ver.h>
BOOL GetFileResource(lpszFileName, lpszResType, lpszResID, dwFileOffset, dwResLen, lpvData)

function GetFileResource(FileName: PChar; ResType: PChar; ResID: PChar; FileOffset: Longint; ResLen: Longint; Data: PChar): Bool;

The **GetFileResource** function copies the specified resource from the specified file into the specified buffer. To obtain the appropriate buffer size, the application can call the **GetFileResourceSize** function before calling **GetFileResource**.

Parameters

lpszFileName	Points to the buffer that contains the name of the file containing the resource.
lpszResType	Points to a value that is created by using the **MAKEINTRESOURCE** macro with the numbered resource type. This value is typically VS_FILE_INFO.
lpszResID	Points to a value that is created by using the **MAKEINTRESOURCE** macro with the numbered resource identifier. This value is typically VS_VERSION_INFO.

dwFileOffset	Specifies the offset of the resource within the file. The **GetFileResourceSize** function returns this value. If this parameter is NULL, the **GetFileResource** function searches the file for the resource.
dwResLen	Specifies the buffer size, in bytes, identified by the *lpvData* parameter. The **GetFileResourceSize** function returns the buffer size required to hold the resource. If the buffer is not large enough, the resource data is truncated to the size of the buffer.
lpvData	Points to the buffer that will receive a copy of the resource. If the buffer is not large enough, the resource data is truncated.

Return Value The return value is nonzero if the function is successful. Otherwise, it is zero, indicating the function could not find the file, could not find the resource, or produced an MS-DOS error. The **GetFileResource** function returns no information about the type of error that occurred.

Comments If the *dwFileOffset* parameter is zero, the **GetFileResource** function determines the location of the resource by using the *lpszResType* and *lpszResID* parameters.

If *dwFileOffset* is not zero, **GetFileResource** assumes that *dwFileOffset* is the return value of **GetFileResourceSize** and, therefore, ignores *lpszResType* and *lpszResID*.

See Also **GetFileResourceSize**

GetFileResourceSize 3.1

Syntax

```
#include <ver.h>
DWORD GetFileResourceSize(lpszFileName, lpszResType, lpszResID,
lpdwFileOffset)
```

```
function GetFileResourceSize(FileName: PChar; ResType: PChar; ResID:
PChar; var FileOffset: Longint): Longint;
```

The **GetFileResourceSize** function searches the specified file for the resource of the specified type and identifier.

Parameters

lpszFileName	Points to the buffer that contains the name of the file in which to search for the resource.

lpszResType	Points to a value that is created by using the **MAKEINTRESOURCE** macro with the numbered resource type. This value is typically VS_FILE_INFO.
lpszResID	Points to a value that is created by using the **MAKEINTRESOURCE** macro with the numbered resource identifier. This value is typically VS_VERSION_INFO.
lpdwFileOffset	Points to a 16-bit value that the **GetFileResourceSize** function fills with the offset to the resource within the file.

Return Value The return value is the size of the resource, in bytes. The return value is NULL if the function could not find the file, the file does not have any resources attached, or the function produced an MS-DOS error. The **GetFileResourceSize** function returns no information about the type of error that occurred.

See Also **GetFileResource**

GetFileTitle 3.1

Syntax

```
#include <commdlg.h>
int GetFileTitle(lpszFile, lpszTitle, cbBuf)
```

```
function GetFileTitle(FileName, Title: PChar; TitleSize: Word): Integer;
```

The **GetFileTitle** function returns the title of the file identified by the *lpszFile* parameter.

Parameters

lpszFile	Points to the name and location of an MS-DOS file.
lpszTitle	Points to a buffer into which the function is to copy the name of the file.
cbBuf	Specifies the length, in bytes, of the buffer to which the *lpszTitle* parameter points.

Return Value The return value is zero if the function is successful. The return value is a negative number if the filename is invalid. The return value is a positive integer that specifies the required buffer size, in bytes, if the buffer to which the *lpszTitle* parameter points is too small.

Comments The function returns an error value if the buffer pointed to by the *lpszFile* parameter contains any of the following:

- An empty string
- A string containing a wildcard (*), opening bracket ([), or closing bracket (])
- A string that ends with a colon (:), slash mark (/), or backslash (\)
- A string whose length exceeded the length of the buffer
- An invalid character (for example, a space or unprintable character).

The required buffer size includes the terminating null character.

GetFileVersionInfo 3.1

Syntax

```
#include <ver.h>
BOOL GetFileVersionInfo(lpszFileName, handle, cbBuf, lpvData)
```

```
function GetFileVersionInfo(FileName: PChar; Handle: Longint; Len:
Longint; Data: PChar): Bool;
```

The **GetFileVersionInfo** function returns version information about the specified file. The application must call the **GetFileVersionInfoSize** function before calling **GetFileVersionInfo** to obtain the appropriate handle if the handle is not NULL.

Parameters

lpszFileName	Points to the buffer that contains the name of the file.
handle	Identifies the file-version information. The **GetFileVersionInfoSize** function returns this handle, or it may be NULL. If the *handle* parameter is NULL, the **GetFileVersionInfo** function searches the file for the version information.
cbBuf	Specifies the buffer size, in bytes, identified by the *lpvData* parameter. The **GetFileVersionInfoSize** function returns the buffer size required to hold the file-version information. If the buffer is not large enough, the file-version information is truncated to the size of the buffer.
lpvData	Points to the buffer that will receive the file-version information. This parameter is used by a subsequent call to the **VerQueryValue** function.

Return Value

The return value is nonzero if the function is successful. Otherwise, it is zero, indicating the file does not exist or the *handle* parameter is invalid. The **GetFileVersionInfo** function returns no information about the type of error that occurred.

Comments The file version information is organized in a **VS_VERSION_INFO** block.

Currently, the **GetFileVersionInfo** function recognizes only version-information created by Microsoft Resource Compiler (RC).

See Also **GetFileVersionInfoSize, VerQueryValue**

GetFileVersionInfoSize 3.1

Syntax

```
#include <ver.h>
DWORD GetFileVersionInfoSize(lpszFileName, lpdwHandle)
```

```
function GetFileVersionInfoSize(FileName: PChar; var Handle: Longint):
Longint;
```

The **GetFileVersionInfoSize** function determines whether it can obtain version information from the specified file. If version information is available, **GetFileVersionInfoSize** returns the size of the buffer required to hold the version information. It also returns a handle that can be used in a subsequent call to the **GetFileVersionInfo** function.

Parameters

lpszFileName Points to the buffer that contains the name of the file.

lpdwHandle Points to a 32-bit value that the **GetFileVersionInfoSize** function fills with the handle to the file-version information. The **GetFileVersionInfo** function can use this handle.

Return Value The return value is the buffer size, in bytes, required to hold the version information if the function is successful. The return value is NULL if the function could not find the file, could not find the version information, or produced an MS-DOS error. The **GetFileVersionInfoSize** function returns no information about the type of error that occurred.

Comments The file version information is organized in a **VS_VERSION_INFO** block.

See Also **GetFileVersionInfo**

GetFontData 3.1

Syntax

```
DWORD GetFontData(hdc, dwTable, dwOffset, lpvBuffer, cbData)
```

```
function GetFontData(hdc: HDC; dwTable, dwOffset: Longint; lpvBuffer:
PChar; cbData: Longint): Longint;
```

The **GetFontData** function retrieves font-metric information from a scalable font file. The information to retrieve is identified by specifying an offset into the font file and the length of the information to return.

Parameters

hdc — Identifies the device context.

dwTable — Specifies the name of the metric table to be returned. This parameter can be one of the metric tables documented in the TrueType Font Files specification, published by Microsoft Corporation. If this parameter is zero, the information is retrieved starting at the beginning of the font file.

dwOffset — Specifies the offset from the beginning of the table at which to begin retrieving information. If this parameter is zero, the information is retrieved starting at the beginning of the table specified by the *dwTable* parameter. If this value is greater than or equal to the size of the table, **GetFontData** returns zero.

lpvBuffer — Points to a buffer that will receive the font information. If this value is NULL, the function returns the size of the buffer required for the font data specified in the *dwTable* parameter.

cbData — Specifies the length, in bytes, of the information to be retrieved. If this parameter is zero, **GetFontData** returns the size of the data specified in the *dwTable* parameter.

Return Value

The return value specifies the number of bytes returned in the buffer pointed to by the *lpvBuffer* parameter, if the function is successful. Otherwise, it is –1.

Comments

An application can sometimes use the **GetFontData** function to save a TrueType font with a document. To do this, the application determines whether the font can be embedded and then retrieves the entire font file, specifying zero for the *dwTable*, *dwOffset*, and *cbData* parameters.

Applications can determine whether a font can be embedded by checking the **otmfsType** member of the **OUTLINETEXTMETRIC** structure. If bit 1 of **otmfsType** is set, embedding is not permitted for the font. If bit 1 is clear, the font can be embedded. If bit 2 is set, the embedding is read-only.

If an application attempts to use this function to retrieve information for a non-TrueType font, the **GetFontData** function returns –1.

Example The following example retrieves an entire TrueType font file:

```
HGLOBAL hglb;
DWORD dwSize;
void FAR* lpvBuffer;

dwSize = GetFontData(hdc, NULL, 0L, NULL, 0L); /* get file size   */

hglb = GlobalAlloc(GPTR, dwSize);  /* allocate memory */
lpvBuffer = GlobalLock(hglb);
GetFontData(hdc, NULL, 0L, lpvBuffer, dwSize);  /* retrieve data   */
```

The following retrieves an entire TrueType font file 4K at a time:

```
#define SIZE  4096
BYTE Buffer[SIZE];
DWORD dwOffset;
DWORD dwSize;
dwOffset = 0L;
while(dwSize = GetFontData(hdc, NULL, dwOffset, Buffer, SIZE)) {
  .
  . /* process data in buffer */
  .
  dwOffset += dwSize;
}
```

The following example retrieves a TrueType font table:

```
HGLOBAL hglb;
DWORD dwSize;
void FAR* lpvBuffer;

LPSTR lpszTable;
DWORD dwTable;

lpszTable = "cmap";
dwTable   = *(LPDWORD) lpszTable;          /* construct DWORD type */

dwSize = GetFontData(hdc, dwTable, 0L, NULL, 0L); /* get table size */

hglb = GlobalAlloc(GPTR, dwSize);                /* allocate memory */
lpvBuffer = GlobalLock(hglb);
GetFontData(hdc, dwTable, 0L, lpvBuffer, dwSize);  /* retrieve data */
```

See Also **GetOutlineTextMetrics**

GetFreeFileHandles 3.1

Syntax #include <stress.h>
int GetFreeFileHandles(void)

function GetFreeFileHandles: Integer;

The **GetFreeFileHandles** function returns the number of file handles available to the current instance.

Parameters This function has no parameters.

Return Value The return value is the number of file handles available to the current instance.

GetFreeSystemResources 3.1

Syntax UINT GetFreeSystemResources(fuSysResource)

function GetFreeSystemResources(SysResource: Word): Word;

The **GetFreeSystemResources** function returns the percentage of free space for system resources.

Parameters *fuSysResource* Specifies the type of resource to be checked. This parameter can be one of the following values:

Value	Meaning
GFSR_SYSTEMRESOURCES	Returns the percentage of free space for system resources.
GFSR_GDIRESOURCES	Returns the percentage of free space for GDI resources. GDI resources include device-context handles, brushes, pens, regions, fonts, and bitmaps.
GFSR_USERRESOURCES	Returns the percentage of free space for USER resources. These resources include window and menu handles.

Return Value The return value specifies the percentage of free space for resources, if the function is successful.

Comments Since the return value from this function does not guarantee that an application will be able to create a new object, applications should not use this function to determine whether it will be possible to create an object.

See Also **GetFreeSpace**

GetGlyphOutline 3.1

Syntax DWORD GetGlyphOutline(hdc, uChar, fuFormat, lpgm, cbBuffer, lpBuffer, lpmat2)

function GetGlyphOutline(hdc: HDC; uChar, fuFormat: Word; var lpgm: TGlyphMetrics; cbBuffer: Longint; lpBuffer: PChar; var lpmat2: TMat2): Longint;

The **GetGlyphOutline** function retrieves the outline curve or bitmap for an outline character in the current font.

Parameters

hdc Identifies the device context.

uChar Specifies the character for which information is to be returned.

fuFormat Specifies the format in which the function is to return information. It can be one of the following values:

Value	Meaning
GGO_BITMAP	Returns the glyph bitmap. When the function returns, the buffer pointed to by the *lpBuffer* parameter contains a 1-bit-per-pixel bitmap whose rows start on doubleword boundaries.
GGO_NATIVE	Returns the curve data points in the rasterizer's native format, using device units. When this value is specified, any transformation specified in the *lpmat2* parameter is ignored.

When the value of this parameter is zero, the function fills in a **GLYPHMETRICS** structure but does not return glyph-outline data.

lpgm Points to a **GLYPHMETRICS** structure that describes the placement of the glyph in the character cell. The **GLYPHMETRICS** structure has the following form:

```
typedef struct tagGLYPHMETRICS {   /* gm */
    UINT  gmBlackBoxX;
    UINT  gmBlackBoxY;
    POINT gmptGlyphOrigin;
    int   gmCellIncX;
    int   gmCellIncY;
} GLYPHMETRICS;
```

cbBuffer Specifies the size of the buffer into which the function copies information about the outline character. If this value is zero and the *fuFormat* parameter is either the GGO_BITMAP or GGO_NATIVE values, the function returns the required size of the buffer.

lpBuffer Points to a buffer into which the function copies information about the outline character. If the *fuFormat* parameter specifies the GGO_NATIVE value, the information is copied in the form of **TTPOLYGONHEADER** and **TTPOLYCURVE** structures. If this value is NULL and the *fuFormat* parameter is either the GGO_BITMAP or GGO_NATIVE value, the function returns the required size of the buffer.

lpmat2 Points to a **MAT2** structure that contains a transformation matrix for the character. This parameter cannot be NULL, even when the GGO_NATIVE value is specified for the *fuFormat* parameter. The **MAT2** structure has the following form:

```
typedef struct tagMAT2 {   /* mat2 */
    FIXED eM11;
    FIXED eM12;
    FIXED eM21;
    FIXED eM22;
} MAT2;
```

Return Value The return value is the size, in bytes, of the buffer required for the retrieved information if the *cbBuffer* parameter is zero or the *lpBuffer* parameter is NULL. Otherwise, it is a positive value if the function is successful, or –1 if there is an error.

Comments An application can rotate characters retrieved in bitmap format by specifying a 2-by-2 transformation matrix in the structure pointed to by the *lpmat2* parameter.

A glyph outline is returned as a series of contours. Each contour is defined by a **TTPOLYGONHEADER** structure followed by as many **TTPOLYCURVE** structures as are required to describe it. 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).

For example, the **GetGlyphOutline** function retrieves the following information about the lowercase "i" in the Arial TrueType font:

```
dwrc = 88                          /* total size of native buffer     */

TTPOLYGONHEADER #1                 /* contour for dot on i            */
  cb     = 44                      /* size for contour                */
  dwType = 24                      /* TT_POLYGON_TYPE                 */
  pfxStart = 1.000, 11.000

  TTPOLYCURVE #1
    wType  = TT_PRIM_LINE
    cpfx   = 3
    pfx[0] = 1.000, 12.000
    pfx[1] = 2.000, 12.000
    pfx[2] = 2.000, 11.000         /* automatically close to pfxStart */

TTPOLYGONHEADER #2                 /* contour for body of i           */
  cb     = 44
  dwType = 24                      /* TT_POLYGON_TYPE                 */
  pfxStart = 1.000, 0.000

  TTPOLYCURVE #1
    wType  = TT_PRIM_LINE
    cpfx   = 3
    pfx[0] = 1.000, 9.000
    pfx[1] = 2.000, 9.000
    pfx[2] = 2.000, 0.000          /* automatically close to pfxStart */
```

See Also **GetOutlineTextMetrics**

GetKerningPairs 3.1

Syntax int GetKerningPairs(hdc, cPairs, lpkrnpair)

function GetKerningPairs(DC: HDC; i: Integer; Pair: PKerningPair): Integer;

The **GetKerningPairs** function retrieves the character kerning pairs for the font that is currently selected in the specified device context.

Parameters

hdc — Identifies a device context. The **GetKerningPairs** function retrieves kerning pairs for the current font for this device context.

cPairs — Specifies the number of **KERNINGPAIR** structures pointed to by the *lpkrnpair* parameter. The function will not copy more kerning pairs than specified by *cPairs*.

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

```
typedef struct tagKERNINGPAIR {
    WORD wFirst;
    WORD wSecond;
    int  iKernAmount;
} KERNINGPAIR;
```

lpkrnpair Points to an array of **KERNINGPAIR** structures that receive the kerning pairs when the function returns. This array must contain at least as many structures as specified by the *cPairs* parameter. If this parameter is NULL, the function returns the total number of kerning pairs for the font.

Return Value The return value specifies the number of kerning pairs retrieved or the total number of kerning pairs in the font, if the function is successful. It is zero if the function fails or there are no kerning pairs for the font.

GetMessageExtraInfo 3.1

Syntax LONG GetMessageExtraInfo(void)

function GetMessageExtraInfo: Longint;

The **GetMessageExtraInfo** function retrieves the extra information associated with the last message retrieved by the **GetMessage** or **PeekMessage** function. This extra information may be added to a message by the driver for a pointing device or keyboard.

Parameters This function has no parameters.

Return Value The return value specifies the extra information if the function is successful. The meaning of the extra information is device-specific.

See Also **GetMessage, hardware_event, PeekMessage**

GetMsgProc 3.1

Syntax LRESULT CALLBACK GetMsgProc(code, wParam, lParam)

The **GetMsgProc** function is a library-defined callback function that the system calls whenever the **GetMessage** function has retrieved a message from an application queue. The system passes the retrieved message to the callback function before passing the message to the destination window procedure.

Parameters

code
Specifies whether the callback function should process the message or call the **CallNextHookEx** function. If this parameter is less than zero, the callback function should pass the message to **CallNextHookEx** without further processing.

wParam
Specifies a NULL value.

lParam
Points to an **MSG** structure that contains information about the message. The **MSG** structure has the following form:

```
typedef struct tagMSG {      /* msg */
    HWND   hwnd;
    UINT   message;
    WPARAM wParam;
    LPARAM lParam;
    DWORD  time;
    POINT  pt;
} MSG;
```

Return Value The callback function should return zero.

Comments The **GetMsgProc** callback function can examine or modify the message as desired. Once the callback function returns control to the system, the **GetMessage** function returns the message, with any modifications, to the application that originally called it. The callback function does not require a return value.

This callback function must be in a dynamic-link library (DLL).

An application must install the callback function by specifying the WH_GETMESSAGE filter type and the procedure-instance address of the callback function in a call to the **SetWindowsHookEx** function.

GetMsgProc is a placeholder for the library-defined function name. The actual name must be exported by including it in an **EXPORTS** statement in the library's module-definition (.DEF) file.

See Also **CallNextHookEx, GetMessage, SetWindowsHookEx**

GetNextDriver 3.1

Syntax HDRVR GetNextDriver(hdrvr, fdwFlag)

function GetNextDriver(Driver: THandle; lParam: Longint): THandle;

The **GetNextDriver** function enumerates instances of an installable driver.

Parameters

hdrvr
Identifies the installable driver for which instances should be enumerated. This parameter must be retrieved by the **OpenDriver** function. If this parameter is NULL, the enumeration begins at either the beginning or end of the list of installable drivers (depending on the setting of the flags in the *fdwFlag* parameter).

fdwFlag
Specifies whether the function should return a handle identifying only the first instance of a driver and whether the function should return handles identifying the instances of the driver in the order in which they were loaded. This parameter can be one or more of the following flags:

Value	Meaning
GND_FIRSTINSTANCEONLY	Returns a handle identifying the first instance of an installable driver. When this flag is set, the function will enumerate only the first instance of an installable driver, no matter how many times the driver has been installed.
GND_FORWARD	Enumerates subsequent instances of the driver. (Using this flag has the same effect as not using the GND_REVERSE flag.)
GND_REVERSE	Enumerates instances of the driver as it was loaded—each subsequent call to the function returns the handle of the next instance.

Return Value

The return value is the instance handle of the installable driver if the function is successful.

GetOpenClipboardWindow 3.1

Syntax

HWND GetOpenClipboardWindow(void)

function GetOpenClipboardWindow: HWnd;

The **GetOpenClipboardWindow** function retrieves the handle of the window that currently has the clipboard open.

Parameters

This function has no parameters.

Return Value

The return value is the handle of the window that has the clipboard open, if the function is successful. Otherwise, it is NULL.

See Also

GetClipboardOwner, GetClipboardViewer, OpenClipboard

GetOpenFileName 3.1

Syntax #include <commdlg.h>
BOOL GetOpenFileName(lpofn)

function GetOpenFileName(var OpenFile: TOpenFilename): Bool;

The **GetOpenFileName** function creates a system-defined dialog box that makes it possible for the user to select a file to open.

Parameters

lpofn Points to an **OPENFILENAME** structure that contains information used to initialize the dialog box. When the **GetOpenFileName** function returns, this structure contains information about the user's file selection.

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

```
#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;
```

Return Value The return value is nonzero if the user selects a file to open. It is zero if an error occurs, if the user chooses the Cancel button, if the user chooses the Close command on the System menu to close the dialog box, or if the buffer identified by the **lpstrFile** member of the **OPENFILENAME** structure is too small to contain the string that specifies the selected file.

Errors The **CommDlgExtendedError** function retrieves the error value, which may be one of the following values:

CDERR_FINDRESFAILURE
CDERR_INITIALIZATION
CDERR_LOCKRESFAILURE
CDERR_LOADRESFAILURE
CDERR_LOADSTRFAILURE
CDERR_MEMALLOCFAILURE
CDERR_MEMLOCKFAILURE
CDERR_NOHINSTANCE
CDERR_NOHOOK
CDERR_NOTEMPLATE
CDERR_STRUCTSIZE
FNERR_BUFFERTOOSMALL
FNERR_INVALIDFILENAME
FNERR_SUBCLASSFAILURE

Comments If the hook function (to which the **lpfnHook** member of the **OPENFILENAME** structure points) processes the WM_CTLCOLOR message, this function must return a handle of the brush that should be used to paint the control background.

Example The following example copies file-filter strings into a buffer, initializes an **OPENFILENAME** structure, and then creates an Open dialog box.

The file-filter strings are stored in the resource file in the following form:

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

The replaceable character at the end of the string is used to break the entire string into separate strings, while still guaranteeing that all the strings are continguous in memory.

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

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

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

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

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

/* Set all structure members to zero. */

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

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

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

See Also **GetSaveFileName**

GetOutlineTextMetrics 3.1

Syntax WORD GetOutlineTextMetrics(hdc, cbData, lpotm)

function GetOutlineTextMetrics(hdc: HDC; cbData: Word; var lpotm: TOutlineTextMetric): Word;

The **GetOutlineTextMetrics** function retrieves metric information for TrueType fonts.

Parameters

hdc	Identifies the device context.
cbData	Specifies the size, in bytes, of the buffer to which information is returned.

lpotm Points to an **OUTLINETEXTMETRIC** structure. If this parameter is NULL, the function returns the size of the buffer required for the retrieved metric information. The **OUTLINETEXTMETRIC** structure has the following form:

```
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;
```

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

Comments The **OUTLINETEXTMETRIC** structure contains most of the font metric information provided with the TrueType format, including a **TEXTMETRIC** structure. The last four members of the **OUTLINETEXTMETRIC** structure are pointers to strings. Applications should allocate space for these strings in addition to the space required for the other members. Because there is no system-imposed limit to the size of the strings, the simplest method for allocating memory is to retrieve the required size by specifying NULL for the *lpotm* parameter in the first call to the **GetOutlineTextMetrics** function.

See Also **GetTextMetrics**

GetQueueStatus 3.1

Syntax DWORD GetQueueStatus(fuFlags)

function GetQueueStatus(Flags: Word): Longint;

The **GetQueueStatus** function returns a value that indicates the type of messages in the queue.

This function is very fast and is typically used inside speed-critical loops to determine whether the **GetMessage** or **PeekMessage** function should be called to process input.

GetQueueStatus returns two sets of information: whether any new messages have been added to the queue since **GetQueueStatus**, **GetMessage**, or **PeekMessage** was last called, and what kinds of events are currently in the queue.

Parameters *fuFlags* Specifies the queue-status flags to be retrieved. This parameter can be a combination of the following values:

Value	Meaning
QS_KEY	WM_CHAR message is in the queue.
QS_MOUSE	WM_MOUSEMOVE or WM_*BUTTON* message is in the queue.
QS_MOUSEMOVE	WM_MOUSEMOVE message is in the queue.
QS_MOUSEBUTTON	WM_*BUTTON* message is in the queue.
QS_PAINT	WM_PAINT message is in the queue.
QS_POSTMSG	Posted message other than those listed above is in the queue.
QS_SENDMSG	Message sent by another application is in the queue.
QS_TIMER	WM_TIMER message is in the queue.

Return Value The high-order word of the return value indicates the types of messages currently in the queue. The low-order word shows the types of messages added to the queue and are still in the queue since the last call to the **GetQueueStatus**, **GetMessage**, or **PeekMessage** function.

Comments The existence of a QS_ flag in the return value does not guarantee that a subsequent call to the **PeekMessage** or **GetMessage** function will return a message. **GetMessage** and **PeekMessage** perform some internal filtering computation that may cause the message to be processed internally. For this reason, the return value from **GetQueueStatus** should be considered only a hint as to whether **GetMessage** or **PeekMessage** should be called.

See Also **GetInputState, GetMessage, PeekMessage**

GetRasterizerCaps 3.1

Syntax BOOL GetRasterizerCaps(lpraststat, cb)

function GetRasterizerCaps(var lpraststat: TRasterizer_Status; cb: Integer): Bool;

The **GetRasterizerCaps** function returns flags indicating whether TrueType fonts are installed in the system.

Parameters

lpraststat Points to a **RASTERIZER_STATUS** structure that receives information about the rasterizer. The **RASTERIZER_STATUS** structure has the following form:

```
typedef struct  tagRASTERIZER_STATUS {    /* rs */
    int   nSize;
    int   wFlags;
    int   nLanguageID;
} RASTERIZER_STATUS;
```

cb Specifies the number of bytes that will be copied into the structure pointed to by the *lpraststat* parameter.

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

Comments The **GetRasterizerCaps** function enables applications and printer drivers to determine whether TrueType is installed.

If the TT_AVAILABLE flag is set in the **wFlags** member of the **RASTERIZER_STATUS** structure, at least one TrueType font is installed. If the TT_ENABLED flag is set, TrueType is enabled for the system.

See Also **GetOutlineTextMetrics**

GetSaveFileName 3.1

Syntax #include <commdlg.h>
BOOL GetSaveFileName(lpofn)

function GetSaveFileName(var OpenFile: TOpenFilename): Bool;

The **GetSaveFileName** function creates a system-defined dialog box that makes it possible for the user to select a file to save.

Parameters

lpofn Points to an **OPENFILENAME** structure that contains information used to initialize the dialog box. When the **GetSaveFileName** function returns, this structure contains information about the user's file selection.

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

```
#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;
} PENFILENAME;
```

Return Value The return value is nonzero if the user selects a file to save. It is zero if an error occurs, if the user clicks the Cancel button, if the user chooses the Close command on the System menu to close the dialog box, or if the buffer identified by the **lpstrFile** member of the **OPENFILENAME** structure is too small to contain the string that specifies the selected file.

Errors The **CommDlgExtendedError** retrieves the error value, which may be one of the following values:

CDERR_FINDRESFAILURE
CDERR_INITIALIZATION
CDERR_LOCKRESFAILURE
CDERR_LOADRESFAILURE
CDERR_LOADSTRFAILURE
CDERR_MEMALLOCFAILURE
CDERR_MEMLOCKFAILURE
CDERR_NOHINSTANCE
CDERR_NOHOOK
CDERR_NOTEMPLATE
CDERR_STRUCTSIZE
FNERR_BUFFERTOOSMALL
FNERR_INVALIDFILENAME
FNERR_SUBCLASSFAILURE

Comments If the hook function (to which the **lpfnHook** member of the **OPENFILENAME** structure points) processes the WM_CTLCOLOR message, this function must return a handle for the brush that should be used to paint the control background.

Example The following example copies file-filter strings (filename extensions) into a buffer, initializes an **OPENFILENAME** structure, and then creates a Save As dialog box.

The file-filter strings are stored in the resource file in the following form:

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

The replaceable character at the end of the string is used to break the entire string into separate strings, while still guaranteeing that all the strings are continguous in memory.

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

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

```
 */
GetSystemDirectory(szDirName, sizeof(szDirName));

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

chReplace = szFilter[cbString - 1]; /* retrieve wild character */

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

/* Set all structure members to zero. */

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

/* Initialize the OPENFILENAME members. */

szFile[0] = '\0';

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

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

See Also **GetOpenFileName**

GetSelectorBase 3.1

Syntax DWORD GetSelectorBase(uSelector)

function GetSelectorBase(Selector: Word): Longint;

The **GetSelectorBase** function retrieves the base address of a selector.

Parameters *uSelector* Specifies the selector whose base address is retrieved.

Return Value This function returns the base address of the specified selector.

See Also **GetSelectorLimit, SetSelectorBase, SetSelectorLimit**

GetSelectorLimit 3.1

Syntax DWORD GetSelectorLimit(uSelector)

function GetSelectorLimit(Selector: Word): Longint;

The **GetSelectorLimit** function retrieves the limit of a selector.

Parameters *uSelector* Specifies the selector whose limit is being retrieved.

Return Value This function returns the limit of the specified selector.

See Also **GetSelectorBase, SetSelectorBase, SetSelectorLimit**

GetSystemDebugState 3.1

Syntax LONG GetSystemDebugState(void)

function GetSystemDebugState: Longint;

The **GetSystemDebugState** function retrieves information about the state of the system. A Windows-based debugger can use this information to determine whether to enter hard mode or soft mode upon encountering a breakpoint.

Parameters This function has no parameters.

Return Value The return value can be one or more of the following values:

Value	Meaning
SDS_MENU	Menu is displayed.
SDS_SYSMODAL	System-modal dialog box is displayed.
SDS_NOTASKQUEUE	Application queue does not exist yet and, therefore, the application cannot accept posted messages.
SDS_DIALOG	Dialog box is displayed.
SDS_TASKISLOCKED	Current task is locked and, therefore, no other task is permitted to run.

GetSystemDir 3.1

Syntax

#include <ver.h>
UINT GetSystemDir(lpszWinDir, lpszBuf, cbBuf)

function GetSystemDir(AppDir: PChar; Buffer: PChar; Size: Integer): Word;

The **GetSystemDir** function retrieves the path of the Windows system directory. This directory contains such files as Windows libraries, drivers, and fonts.

GetSystemDir is used by MS-DOS applications that set up Windows applications; it exists only in the static-link version of the File Installation library. Windows applications should use the **GetSystemDirectory** function to determine the Windows directory.

Parameters

lpszWinDir	Points to the Windows directory retrieved by a previous call to the **GetWindowsDir** function.
lpszBuf	Points to the buffer that is to receive the null-terminated string containing the path.
cbBuf	Specifies the size, in bytes, of the buffer pointed to by the *lpszBuf* parameter.

Return Value

The return value is the length of the string copied to the buffer, in bytes, including the terminating null character, if the function is sucessful. If the return value is greater than the *cbBuf* parameter, the return value is the size of the buffer required to hold the path. The return value is zero if the function fails.

Comments

An application must call the **GetWindowsDir** function before calling the **GetSystemDir** function to obtain the correct *lpszWinDir* value.

The path that this function retrieves does not end with a backslash unless the Windows system directory is the root directory. For example, if the system directory is named WINDOWS\SYSTEM on drive C, the path of the system directory retrieved by this function is C:\WINDOWS\SYSTEM.

See Also

GetSystemDirectory, GetWindowsDir

GetTextExtentPoint 3.1

Syntax BOOL GetTextExtentPoint(hdc, lpszString, cbString, lpSize)

function GetTextExtentPoint(DC: HDC; Str: PChar; Count: Integer; var Size: Integer): Bool;

The **GetTextExtentPoint** function computes the width and height of the specified text string. The **GetTextExtentPoint** function uses the currently selected font to compute the dimensions of the string. The width and height, in logical units, are computed without considering any clipping.

The **GetTextExtentPoint** function may be used as either a wide-character function (where text arguments must use Unicode) or an ANSI function (where text arguments must use characters from the Windows 3.*x* character set

Parameters

hdc	Identifies the device context.
lpszString	Points to a text string.
cbString	Specifies the number of bytes in the text string.
lpSize	Points to a **SIZE** structure that will receive the dimensions of the string The **SIZE** structure has the following form:

```
typedef struct tagSIZE {
    int cx;
    int cy;
} SIZE;
```

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

Comments Because some devices do not place characters in regular cell arrays—that is, because they carry out kerning—the sum of the extents of the characters in a string may not be equal to the extent of the string.

The calculated width takes into account the intercharacter spacing set by the **SetTextCharacterExtra** function.

See Also **SetTextCharacterExtra**

GetTimerResolution 3.1

Syntax DWORD GetTimerResolution(void)

function GetTimerResolution: Longint;

The **GetTimerResolution** function retrieves the number of microseconds per timer tick.

Parameters This function has no parameters.

Return Value The return value is the number of microseconds per timer tick.

See Also **GetTickCount, SetTimer**

GetViewportExtEx 3.1

Syntax BOOL GetViewportExtEx(hdc, lpSize)

function GetViewportExtEx(DC: HDC; Size: PSize): Bool;

The **GetViewportExtEx** function retrieves the x- and y-extents of the device context's viewport.

Parameters

hdc	Identifies the device context.
lpSize	Points to a **SIZE** structure. The x- and y-extents (in device units) are placed in this structure.

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

GetViewportOrgEx 3.1

Syntax BOOL GetViewportOrgEx(hdc, lpPoint)

function GetViewportOrgEx(DC: HDC; Point: PPoint): Bool;

The **GetViewportOrgEx** function retrieves the x- and y-coordinates of the origin of the viewport associated with the specified device context.

Parameters

hdc	Identifies the device context.
lpPoint	Points to a **POINT** structure. The origin of the viewport (in device coordinates) is placed in this structure.

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

GetWinDebugInfo 3.1

Syntax BOOL GetWinDebugInfo(lpwdi, flags)

function GetWinDebugInfo(DebugInfo: PWinDebugInfo; Flags: Word): Bool;

The **GetWinDebugInfo** function retrieves current system-debugging information for the debugging version of the Windows 3.1 operating system.

Parameters

lpwdi Points to a **WINDEBUGINFO** structure that is filled with debugging information. The **WINDEBUGINFO** structure has the following form:

```
typedef struct tagWINDEBUGINFO {
    UINT    flags;
    DWORD   dwOptions;
    DWORD   dwFilter;
    char    achAllocModule[8];
    DWORD   dwAllocBreak;
    DWORD   dwAllocCount;
} WINDEBUGINFO;
```

flags Specifies which members of the **WINDEBUGINFO** structure should be filled in. This parameter can be one or more of the following values:

Value	Meaning
WDI_OPTIONS	Fill in the **dwOptions** member of **WINDEBUGINFO**.
WDI_FILTER	Fill in the **dwFilter** member of **WINDEBUGINFO**.
WDI_ALLOCBREAK	Fill in the **achAllocModule**, **dwAllocBreak**, and **dwAllocCount** members of **WINDEBUGINFO**.

Return Value The return value is nonzero if the function is successful. It is zero if the pointer specified in the *lpwdi* parameter is invalid or if the function is not called in the debugging version of Windows 3.1.

Comments The **flags** member of the returned **WINDEBUGINFO** structure is set to the values supplied in the *flags* parameter of this function.

See Also **SetWinDebugInfo**

GetWindowExtEx 3.1

Syntax BOOL GetWindowExtEx(hdc, lpSize)

function GetWindowExtEx(DC: HDC; Size: PSize): Bool;

The **GetWindowExtEx** function retrieves the x- and y-extents of the window associated with the specified device context.

Parameters

hdc Identifies the device context.

lpSize Points to a **SIZE** structure. The x- and y-extents (in logical units) are placed in this structure.

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

GetWindowOrgEx 3.1

Syntax BOOL GetWindowOrgEx(hdc, lpPoint)

function GetWindowOrgEx(DC: HDC; Point: PPoint): Bool;

This function retrieves the x- and y-coordinates of the origin of the window associated with the specified device context.

Parameters

hdc Identifies the device context.

lpPoint Points to a **POINT** structure. The origin of the window (in logical coordinates) is placed in this structure.

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

GetWindowPlacement 3.1

Syntax BOOL GetWindowPlacement(hwnd, lpwndpl)

function GetWindowPlacement(Wnd: HWnd; Placement: PWindowPlacement): Bool;

The **GetWindowPlacement** function retrieves the show state and the normal (restored), minimized, and maximized positions of a window.

Parameters

hwnd — Identifies the window.

lpwndpl — Points to the **WINDOWPLACEMENT** structure that receives the show state and position information. The **WINDOWPLACEMENT** structure has the following form:

```
typedef struct tagWINDOWPLACEMENT {     /* wndpl */
    UINT  length;
    UINT  flags;
    UINT  showCmd;
    POINT ptMinPosition;
    POINT ptMaxPosition;
    RECT  rcNormalPosition;
} WINDOWPLACEMENT;
```

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

See Also **SetWindowPlacement**

GetWindowsDir 3.1

Syntax #include <ver.h>
UINT GetWindowsDir(lpszAppDir, lpszPath, cbPath)

function GetWindowsDir(AppDir: PChar; Buffer: PChar; Size: Integer): Word;

The **GetWindowsDir** function retrieves the path of the Windows directory. This directory contains such files as Windows applications, initialization files, and help files.

GetWindowsDir is used by MS-DOS applications that set up Windows applications; it exists only in the static-link version of the File Installation library. Windows applications should use the **GetWindowsDirectory** function to determine the Windows directory.

Parameters

lpszAppDir — Specifies the current directory in a search for Windows files. If the Windows directory is not on the path, the application must prompt the user for its location and pass that string to the **GetWindowsDir** function in the *lpszAppDir* parameter.

lpszPath — Points to the buffer that will receive the null-terminated string containing the path.

cbPath — Specifies the size, in bytes, of the buffer pointed to by the *lpszPath* parameter.

Return Value The return value is the length of the string copied to the *lpszPath* parameter, including the terminating null character, if the function is successful. If the return value is greater than the *cbPath* parameter, it is the size of the buffer required to hold the path. The return value is zero if the function fails.

Comments The path that this function retrieves does not end with a backslash unless the Windows directory is the root directory. For example, if the Windows directory is named WINDOWS on drive C, the path retrieved by this function is C:\WINDOWS. If Windows is installed in the root directory of drive C, the path retrieved is C:\ .

After the **GetWindowsDir** function locates the Windows directory, it caches the location for use by subsequent calls to the function.

See Also **GetSystemDir, GetWindowsDirectory**

GetWinMem32Version 3.0

Syntax

```
#include &lt;winmem32.h&gt;
WORD GetWinMem32Version(void)
```

```
function GetWinMem32Version: Word;
```

The **GetWinMem32Version** function retrieves the application programming interface (API) version implemented by the WINMEM32.DLL dynamic-link library. This is not the version number of the library itself.

Parameters This function has no parameters.

Return Value The return value specifies the version of the 32-bit memory API implemented by WINMEM32.DLL. The high-order 8 bits contain the major version number, and the low-order 8 bits contain the minor version number.

Global16PointerAlloc 3.0

Syntax #include <winmem32.h>
WORD Global16PointerAlloc(wSelector, dwOffset, lpBuffer, dwSize, wFlags)

function Global16PointerAlloc(Selector: Word; dwOffset: Longint; lpBuffer: PLongint; dwSize: Longint; wFlags: Word): Word;

The **Global16PointerAlloc** function converts a 16:32 pointer into a 16:16 pointer alias that the application can pass to a Windows function or to other 16:16 functions.

Parameters

wSelector	Specifies the selector of the object for which an alias is to be created. This must be the selector returned by a previous call to the **Global32Alloc** function.
dwOffset	Specifies the offset of the first byte for which an alias is to be created. The offset is from the first byte of the object specified by the *wSelector* parameter. Note that *wSelector:dwOffset* forms a 16:32 address of the first byte of the region for which an alias is to be created.
lpBuffer	Points to a four-byte location in memory that receives the 16:16 pointer alias for the specified region.
dwSize	Specifies the addressable size, in bytes, of the region for which an alias is to be created. This value must be no larger than 64K.
wFlags	Reserved; must be zero.

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

WM32_Insufficient_Mem
WM32_Insufficient_Sels
WM32_Invalid_Arg
WM32_Invalid_Flags
WM32_Invalid_Func

Comments When this function returns successfully, the location pointed to by the *lpBuffer* parameter contains a 16:16 pointer to the first byte of the region. This is the same byte to which *wSelector:dwOffset* points.

The returned selector identifies a descriptor for a data segment that has the following attributes: read-write, expand up, and small (B bit clear). The descriptor privilege level (DPL) and the granularity (the G bit) are set at the system's discretion, so you should make no assumptions regarding their settings. The DPL and requestor privilege level (RPL) are appropriate for a Windows application.

An application must not change the setting of any bits in the DPL or the RPL selector. Doing so can result in a system crash and will prevent the application from running on compatible platforms.

Because of tiling schemes implemented by some systems, the offset portion of the returned 16:16 pointer is not necessarily zero.

When writing your application, you should not assume the size limit of the returned selector. Instead, assume that at least *dwSize* bytes can be addressed starting at the 16:16 pointer created by this function.

See Also **Global16PointerFree**

Global16PointerFree 3.0

Syntax

```
#include <winmem32.h>
WORD Global16PointerFree(wSelector, dwAlias, wFlags)
```

```
function Global16PointerFree(wSelector: Word; dwAlias: Longint;
wFlags: Word): Word;
```

The **Global16PointerFree** function frees the 16:16 pointer alias previously created by a call to the **Global16PointerAlloc** function.

Parameters

wSelector	Specifies the selector of the object for which the alias is to be freed. This must be the selector returned by a previous call to the **Global32Alloc** function.
dwAlias	Specifies the 16:16 pointer alias to be freed. This must be the alias (including the original offset) returned by a previous call to the **Global16PointerAlloc** function.
wFlags	Reserved; must be zero.

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

WM32_Insufficient_Mem
WM32_Insufficient_Sels
WM32_Invalid_Arg

WM32_Invalid_Flags
WM32_Invalid_Func

Comments An application should free a 16:16 pointer alias as soon as it is no longer needed. Freeing the alias releases space in the descriptor table, a limited system resource.

See Also **Global16PointerAlloc**

Global32Alloc 3.0

Syntax #include <winmem32.h>
WORD Global32Alloc(dwSize, lpSelector, dwMaxSize, wFlags)

function Global32Alloc(dwSize: Longint; lpSelector: PWord; dwMaxSize, wFlags :Word): Word;

The **Global32Alloc** function allocates a memory object to be used as a 16:32 (USE32) code or data segment and retrieves the selector portion of the 16:32 address of the memory object. The first byte of the object is at offset 0 from this selector.

Parameters

dwSize	Specifies the initial size, in bytes, of the object to be allocated. This value must be in the range 1 through (16 megabytes – 64K).
lpSelector	Points to a 2-byte location in memory that receives the selector portion of the 16:32 address of the allocated object.
dwMaxSize	Specifies the maximum size, in bytes, that the object will reach when it is reallocated by the **Global32Realloc** function. This value must be in the range 1 through (16 megabytes – 64 K). If the application will never reallocate this memory object, the *dwMaxSize* parameter should be set to the same value as the *dwSize* parameter.
wFlags	Depends on the return value of the **GetWinMem32Version** function. If the return value is less than 0x0101, this parameter must be zero. If the return value is greater than or equal to 0x0101, this parameter can be set to GMEM_DDESHARE (to make the object shareable). Otherwise, this parameter should be zero. For more information about GMEM_DDESHARE, see the description of the **GlobalAlloc** function.

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

WM32_Insufficient_Mem
WM32_Insufficient_Sels
WM32_Invalid_Arg
WM32_Invalid_Flags
WM32_Invalid_Func

Comments If the **Global32Alloc** function fails, the value to which the *lpSelector* parameter points is zero. If the function succeeds, *lpSelector* points to the selector of the object. The valid range of offsets for the object referenced by this selector is 0 through (but not including) *dwSize*.

In Windows 3.0 and later, the largest object that can be allocated is 0x00FF0000 (16 megabytes – 64K). This is the limitation placed on WINMEM32.DLL by the current Windows kernel.

The returned selector identifies a descriptor for a data segment that has the following attributes: read-write, expand-up, and big (B bit set). The descriptor privilege level (DPL) and the granularity (the G bit) are set at the system's discretion, so you should make no assumptions regarding these settings. Because the system sets the granularity, the size of the object (and the selector size limit) may be greater than the requested size by up to 4095 bytes (4K minus 1). The DPL and requestor privilege level (RPL) will be appropriate for a Windows application.

An application must not change the setting of any bits in the DPL or the RPL selector. Doing so can result in a system crash and will prevent the application from running on compatible platforms.

The allocated object is neither movable nor discardable but can be paged. An application should not page-lock a 32-bit memory object. Page-locking an object is useful only if the object contains code or data that is used at interrupt time, and 32-bit memory cannot be used at interrupt time.

See Also **Global32Free, Global32Realloc**

Global32CodeAlias 3.0

Syntax

#include <winmem32.h>
WORD Global32CodeAlias(wSelector, lpAlias, wFlags)

function Global32CodeAlias(wSelector: Word; lpAlias: PLongint; wFlags: Word): Word;

The **Global32CodeAlias** function creates a 16:32 (USE32) code-segment alias selector for a 32-bit memory object previously created by the **Global32Alloc** function. This allows the application to execute code contained in the memory object.

Parameters

wSelector	Specifies the selector of the object for which an alias is to be created. This must be the selector returned by a previous call to the **Global32Alloc** function.
lpAlias	Points to a 2-byte location in memory that receives the selector portion of the 16:32 code-segment alias for the specified object.
wFlags	Reserved; must be zero.

Return Value

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

WM32_Insufficient_Mem
WM32_Insufficient_Sels
WM32_Invalid_Arg
WM32_Invalid_Flags
WM32_Invalid_Func

Comments

If the function fails, the value pointed to by the *lpAlias* parameter is zero. If the function is successful, *lpAlias* points to a USE32 code-segment alias for the object specified by the *wSelector* parameter. The first byte of the object is at offset 0 from the selector returned in *lpAlias*. Valid offsets are determined by the size of the object as set by the most recent call to the **Global32Alloc** or **Global32Realloc** function.

The returned selector identifies a descriptor for a code segment that has the following attributes: read-execute, nonconforming, and USE32 (D bit set). The descriptor privilege level (DPL) and the granularity (the G bit) are set at the system's discretion, so you should make no assumptions regarding their settings. The granularity will be consistent with the current data selector for the object. The DPL and requestor privilege level (RPL) are appropriate for a Windows application.

An application must not change the setting of any bits in the DPL or the RPL selector. Doing so can result in a system crash and will prevent the application from running on compatible platforms.

An application should not call this function more than once for an object. Depending on the system, the function might fail if an application calls it a second time for a given object without first calling the **Global32CodeAliasFree** function for the object.

See Also **Global32Alloc, Global32CodeAliasFree**

Global32CodeAliasFree 3.0

Syntax #include <winmem32.h>
WORD Global32CodeAliasFree(wSelector, wAlias, wFlags)

function Global32CodeAliasFree(wSelector, wAlias, wFlags: Word): Word;

The **Global32CodeAliasFree** function frees the 16:32 (USE32) code-segment alias selector previously created by a call to the **Global32CodeAlias** function.

Parameters

wSelector	Specifies the selector of the object for which the alias is to be freed. This must be the selector returned by a previous call to the **Global32Alloc** function.
wAlias	Specifies the USE32 code-segment alias selector to be freed. This must be the alias returned by a previous call to the **Global32CodeAlias** function.
wFlags	Reserved; must be zero.

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

WM32_Insufficient_Mem
WM32_Insufficient_Sels
WM32_Invalid_Arg
WM32_Invalid_Flags
WM32_Invalid_Func

See Also **Global32CodeAlias**

Global32Free 3.0

Syntax #include <winmem32.h>
WORD Global32Free(wSelector, wFlags)

function Global32Free(wSelector, wFlags: Word): Word;

The **Global32Free** function frees an object previously allocated by the **Global32Alloc** function.

Parameters

wSelector	Specifies the selector of the object to be freed. This must be the selector returned by a previous call to the **Global32Alloc** function.
wFlags	Reserved; must be zero.

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

WM32_Insufficient_Mem
WM32_Insufficient_Sels
WM32_Invalid_Arg
WM32_Invalid_Flags
WM32_Invalid_Func

Comments The **Global32Alloc** function frees the object itself; it also frees all aliases created for the object by the 32-bit memory application programming interface (API).

Before terminating, an application must call this function to free each object allocated by the **Global32Alloc** function to ensure that all aliases created for the object are freed.

See Also **Global32Alloc, Global32Realloc**

Global32Realloc 3.0

Syntax #include <winmem32.h>
WORD Global32Realloc(wSelector, dwNewSize, wFlags)

function Global32Realloc(wSelector: Word; swNewSize: Longint; wFlags: Word): Word;

The **Global32Realloc** function changes the size of a 32-bit memory object previously allocated by the **Global32Alloc** function.

Parameters

wSelector	Specifies the selector of the object to be changed. This must be the selector returned by a previous call to the **Global32Alloc** function.
dwNewSize	Specifies the new size, in bytes, of the object. This value must be greater than zero and less than or equal to the size specified by the *dwMaxSize* parameter of the **Global32Alloc** function call that created the object.
wFlags	Reserved; must be zero.

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

WM32_Insufficient_Mem
WM32_Insufficient_Sels
WM32_Invalid_Arg
WM32_Invalid_Flags
WM32_Invalid_Func

Comments If this function fails, the previous state of the object is unchanged. If the function succeeds, it updates the state of the object and the state of all aliases to the object created by the 32-bit memory application programming interface (API) functions. For this reason, an application must call the the **Global32Realloc** function to change the size of the object. Using other Windows functions to manipulate the object results in corrupted aliases.

This function does not change the selector specified by the *wSelector* parameter. If this function succeeds, the new valid range of offsets for the selector is zero through (but not including) *dwNewSize*.

The system determines the appropriate granularity of the object. As a result, the size of the object (and the selector size limit) may be greater than the requested size by up to 4095 bytes (4K minus 1).

See Also **Global32Alloc, Global32Free**

GlobalEntryHandle 3.1

Syntax

#include <toolhelp.h>
BOOL GlobalEntryHandle(lpge, hglb)

function GlobalEntryHandle(lpGlobal: PGlobalEntry; hItem: THandle): Bool;

The **GlobalEntryHandle** function fills the specified structure with information that describes the given global memory object.

Parameters

lpge Points to a **GLOBALENTRY** structure that receives information about the global memory object. The **GLOBALENTRY** structure has the following form:

```
#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;
```

hglb Identifies the global memory object to be described.

Return Value

The return value is nonzero if the function is successful. Otherwise, it is zero. The function fails if the *hglb* value is an invalid handle or selector.

Comments

This function retrieves information about a global memory handle or selector. Debuggers use this function to obtain the segment number of a segment loaded from an executable file.

Before calling the **GlobalEntryHandle** function, an application must initialize the **GLOBALENTRY** structure and specify its size, in bytes, in the **dwSize** member.

See Also

GlobalEntryModule, GlobalFirst, GlobalInfo, GlobalNext

GlobalEntryModule 3.1

Syntax #include <toolhelp.h>
BOOL GlobalEntryModule(lpge, hmod, wSeg)

function GlobalEntryModule(lpGlobal: PGlobalEntry; hModule: THandle; wSeg: Word): Bool;

The **GlobalEntryModule** function fills the specified structure by *lpge* with information about the specified module segment.

Parameters

lpge Points to a **GLOBALENTRY** structure that receives information about the segment specified in the *wSeg* parameter. The **GLOBALENTRY** structure has the following form:

```
#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;
```

hmod Identifies the module that owns the segment.

wSeg Specifies the segment to be described in the **GLOBALENTRY** structure. The number of the first segment in the module is 1. Segment numbers are always contiguous, so if the last valid segment number is 10, all segment numbers 1 through 10 are also valid.

Return Value The return value is nonzero if the function is successful. Otherwise, it is zero. This function fails if the segment in the *wSeg* parameter does not exist in the module specified in the *hmod* parameter.

Comments Debuggers can use the **GlobalEntryModule** function to retrieve global heap information about a specific segment loaded from an executable file.

Typically, the debugger will have symbols that refer to segment numbers; this function translates the segment numbers to heap information.

Before calling **GlobalEntryModule**, an application must initialize the **GLOBALENTRY** structure and specify its size, in bytes, in the **dwSize** member.

See Also **GlobalEntryHandle, GlobalFirst, GlobalInfo, GlobalNext**

GlobalFirst 3.1

Syntax #include <toolhelp.h>
BOOL GlobalFirst(lpge, wFlags)

function GlobalFirst(lpGlobal: PGlobalEntry; wFlags: Word): Bool;

The **GlobalFirst** function fills the specified structure with information that describes the first object on the global heap.

Parameters

lpge Points to a **GLOBALENTRY** structure that receives information about the global memory object. The **GLOBALENTRY** structure has the following form:

```
#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;
```

wFlags Specifies the heap to use. This parameter can be one of the following values:

Value	Meaning
GLOBAL_ALL	Structure pointed to by *lpge* will receive information about the first object on the complete global heap.
GLOBAL_FREE	Structure will receive information about the first object on the free list.
GLOBAL_LRU	Structure will receive information about the first object on the least-recently-used (LRU) list.

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

Comments The **GlobalFirst** function can be used to begin a global heap walk. An application can examine subsequent objects on the global heap by using the **GlobalNext** function. Calls to **GlobalNext** must have the same *wFlags* value as that specified in **GlobalFirst**.

Before calling **GlobalFirst**, an application must initialize the **GLOBALENTRY** structure and specify its size, in bytes, in the **dwSize** member.

See Also **GlobalEntryHandle, GlobalEntryModule, GlobalInfo, GlobalNext**

GlobalHandleToSel 3.1

Syntax #include <toolhelp.h>
WORD GlobalHandleToSel(hglb)

function GlobalHandleToSel(hMem: THandle): Word;

The **GlobalHandleToSel** function converts the given handle to a selector.

Parameters *hglb* Identifies the global memory object to be converted.

Return Value The return value is the selector of the given object if the function is successful. Otherwise, it is zero.

Comments The **GlobalHandleToSel** function converts a global handle to a selector appropriate for Windows, version 3.0 or 3.1, depending on which version is running. A debugging application might use this selector to access a global memory object if the object is not discardable or if the object's attributes are irrelevant.

See Also **GlobalAlloc**

GlobalInfo 3.1

Syntax #include <toolhelp.h>
BOOL GlobalInfo(lpgi)

function GlobalInfo(lpGlobalInfo: PGlobalInfo): Bool;

The **GlobalInfo** function fills the specified structure with information that describes the global heap.

Parameters *lpgi* Points to a **GLOBALINFO** structure that receives information about the global heap. The **GLOBALINFO** structure has the following form:

```
#include <toolhelp.h>

typedef struct tagGLOBALINFO {  /* gi */
    DWORD dwSize;
    WORD  wcItems;
    WORD  wcItemsFree;
    WORD  wcItemsLRU;
} GLOBALINFO;
```

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

Comments The information in the structure can be used to determine how much memory to allocate for a global heap walk.

Before calling the **GlobalInfo** function, an application must initialize the **GLOBALINFO** structure and specify its size, in bytes, in the **dwSize** member.

See Also **GlobalEntryHandle, GlobalEntryModule, GlobalFirst, GlobalNext**

GlobalNext 3.1

Syntax #include <toolhelp.h>
BOOL GlobalNext(lpge, flags)

function GlobalNext(lpGlobal: PGlobalEntry; wFlags: Word): Bool;

The **GlobalNext** function fills the specified structure with information that describes the next object on the global heap.

Parameters

lpge Points to a **GLOBALENTRY** structure that receives information about the global memory object. The **GLOBALENTRY** structure has the following form:

```
#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;
```

flags Specifies heap to use. This parameter can be one of the following values:

Value	Meaning
GLOBAL_ALL	Structure pointed by the *lpge* parameter will receive information about the first object on the complete global heap.
GLOBAL_FREE	Structure will receive information about the first object on the free list.
GLOBAL_LRU	Structure will receive information about the first object on the least-recently-used (LRU) list.

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

Comments The **GlobalNext** function can be used to continue a global heap walk started by the **GlobalFirst**, **GlobalEntryHandle**, or **GlobalEntryModule** functions.

If **GlobalFirst** starts a heap walk, the *flags* value used in **GlobalNext** must be the same as the value used in **GlobalFirst**.

See Also **GlobalEntryHandle, GlobalEntryModule, GlobalFirst, GlobalInfo**

GrayStringProc 2.x

Syntax BOOL CALLBACK GrayStringProc(hdc, lpData, cch)

The **GrayStringProc** function is an application-defined callback function that draws a string as a result of a call to the **GrayString** function.

Parameters

hdc	Identifies a device context with a bitmap of at least the width and height specified by the *cx* and *cy* parameters passed to the **GrayString** function.
lpData	Points to the string to be drawn.
cch	Specifies the length, in characters, of the string.

Return Value The callback function should return TRUE to indicate success. Otherwise it should return FALSE.

Comments The callback function must draw an image relative to the coordinates (0,0).

GrayStringProc is a placeholder for the application-defined function name. The actual name must be exported by including it in an **EXPORTS** statement in the application's module-definition (.DEF) file.

See Also **GrayString**

HardwareProc 3.1

Syntax LRESULT CALLBACK HardwareProc(code, wParam, lParam)

The **HardwareProc** function is an application-defined callback function that the system calls whenever the application calls the **GetMessage** or **PeekMessage** function and there is a hardware event to process. Mouse events and keyboard events are not processed by this hook.

Parameters

code	Specifies whether the callback function should process the message or call the **CallNextHookEx** function. If this value is less than zero, the callback function should pass the message to **CallNextHookEx** without further processing. If this value is HC_NOREMOVE, the application is using the **PeekMessage** function with the PM_NOREMOVE option, and the message will not be removed from the system queue.
wParam	Specifies a NULL value.

lParam Points to a **HARDWAREHOOKSTRUCT** structure. The **HARDWAREHOOKSTRUCT** structure has the following form:

```
typedef struct tagHARDWAREHOOKSTRUCT {  /* hhs */
    HWND    hWnd;
    UINT    wMessage;
    WPARAM  wParam;
    LPARAM  lParam;
} HARDWAREHOOKSTRUCT;
```

Return Value The callback function should return zero to allow the system to process the message; it should be 1 if the message is to be discarded.

Comments This callback function should not install a playback hook because the function cannot use the **GetMessageExtraInfo** function to get the extra information associated with the message.

The callback function must use the Pascal calling convention and must be declared **FAR**. An application must install the callback function by specifying the WH_HARDWARE filter type and the procedure-instance address of the callback function in a call to the **SetWindowsHookEx** function.

HardwareProc is a placeholder for the library-defined function name. The actual name must be exported by including it in an **EXPORTS** statement in the library's module-definition (.DEF) file.

See Also **CallNextHookEx, GetMessageExtraInfo, SetWindowsHookEx**

hardware_event 3.1

```
extrn hardware_event :far

mov   ax, Msg         ; message
mov   cx, ParamL      ; low-order word of lParam of the message
mov   dx, ParamH      ; high-order word of lParam of the message
mov   si, hwnd        ; handle of the destination window
mov   di, wParam      ; wParam of the message
cCall hardware_event
```

The **hardware_event** function places a hardware-related message into the system message queue. This function allows a driver for a non-standard hardware device to place a message into the queue.

Parameters

Msg	Specifies the message to place in the system message queue.
ParamL	Specifies the low-order word of the *lParam* parameter of the message.
lParamH	Specifies the high-order word of the *lParam* parameter of the message.
hwnd	Identifies the window to which the message is directed. This parameter also becomes the low-order word of the *dwExtraInfo* parameter associated with the message. An application can determine the value of this parameter by calling the **GetMessageExtraInfo** function.
wParam	Specifies the *wParam* parameter of the message.

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

Comments An application should not use this function to place keyboard or mouse messages into the system message queue.

An application may only call the **hardware_event** function from an assembly language routine. The application must declare the function as follows:

```
extrn hardware_event :far
```

If the application includes CMACROS.INC, the application can declare the function as follows:

```
extrnFP hardware_event.
```

See Also **GetMessageExtraInfo**

hmemcpy 3.1

Syntax void hmemcpy(hpvDest, hpvSource, cbCopy)

procedure hmemcpy(hpvDest, hpvSource: Pointer; cbCopy: Longint);

The **hmemcpy** function copies bytes from a source buffer to a destination buffer. This function supports huge memory objects (that is, objects larger than 64K, allocated using the **GlobalAlloc** function).

Parameters

hpvDest	Points to a buffer that receives the copied bytes.
hpvSource	Points to a buffer that contains the bytes to be copied.

cbCopy	Specifies the number of bytes to be copied.

Return Value This function does not return a value.

See Also **_hread, _hwrite, lstrcpy**

_hread 3.1

Syntax long _hread(hf, hpvBuffer, cbBuffer)

The **_hread** function reads data from the specified file. This function supports huge memory objects (that is, objects larger than 64K, allocated using the **GlobalAlloc** function).

Parameters

hf	Identifies the file to be read.
hpvBuffer	Points to a buffer that is to receive the data read from the file.
cbBuffer	Specifies the number of bytes to be read from the file.

Return Value The return value indicates the number of bytes that the function read from the file, if the function is successful. If the number of bytes read is less than the number specified in *cbBuffer*, the function reached the end of the file (EOF) before reading the specified number of bytes. The return value is HFILE_ERROR if the function fails.

See Also **_lread, hmemcpy, _hwrite**

_hwrite 3.1

Syntax long _hwrite(hf, hpvBuffer, cbBuffer)

The **_hwrite** function writes data to the specified file. This function supports huge memory objects (that is, objects larger than 64K, allocated using the **GlobalAlloc** function).

Parameters

hf	Identifies the file to be written to.
hpvBuffer	Points to a buffer that contains the data to be written to the file.
cbBuffer	Specifies the number of bytes to be written to the file.

Return Value The return value indicates the number of bytes written to the file, if the function is successful. Otherwise, the return value is HFILE_ERROR.

Comments The buffer specified by *hpvBuffer* cannot extend past the end of a segment.

See Also **hmemcpy, _hread, _lwrite**

InterruptRegister 3.1

Syntax #include <toolhelp.h>
BOOL InterruptRegister(htask, lpfnIntCallback)

function InterruptRegister(hTask: THandle; lpfnIntCallBack: TIntCallBack): Bool;

The **InterruptRegister** function installs a callback function to handle all system interrupts.

Parameters

htask	Identifies the task that is registering the callback function. The *htask* value is for registration purposes, not for filtering interrupts. Typically, this value is NULL, indicating the current task. The only time this value is not NULL is when an application requires more than one interrupt handler.
lpfnIntCallback	Points to the interrupt callback function that will handle interrupts. The Tool Helper library calls this function whenever a task receives an interrupt. The *lpfnIntCallback* value is normally the return value of a call to the **MakeProcInstance** function. This causes the interrupt callback function to be entered with the AX register set to the selector of the application's data segment. Usually, an exported function prolog contains the following code: `mov ds,ax`

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

Comments The syntax of the function pointed to by *lpfnIntCallback* is as follows:

void InterruptRegisterCallback(void)

TIntCallBack = procedure;

InterruptRegisterCallback is a placeholder for the application-defined function name. The actual name must be exported by including it an **EXPORTS** in the application's module-definition file.

An interrupt callback function must be reentrant, must be page-locked, and must explicitly preserve all register values. When the Tool Helper library calls the function, the stack will be organized as shown in the following illustration.

The SS and SP values will not be on the stack unless a low-stack fault occurred. This fault is indicated by the high bit of the interrupt number being set.

When Windows calls a callback function, the AX register contains the DS value for the instance of the application that contains the callback function. For more information about this process, see the **MakeProcInstance** function.

Typically, an interrupt callback function is exported. If it is not exported, the developer should verify that the appropriate stack frame is generated, including the correct DS value.

An interrupt callback function must save and restore all register values. The function must also do one of the following:

- Execute an **retf** instruction if it does not handle the interrupt. The Tool Helper library will pass the interrupt to the next appropriate handler in the interrupt handler list.
- Terminate the application by using the **TerminateApp** function.
- Correct the problem that caused the interrupt, clear the first 10 bytes of the stack, and execute an **iret** instruction. This action will restart execution at the specified address. An application may change this address, if necessary.
- Execute a nonlocal goto to a known position in the application by using the **Catch** and **Throw** functions. This type of interrupt handling can be hazardous; the system may be in an unstable state and another fault may occur. Applications that handle interrupts in this way must verify that the fault was a result of the application's code.

The Tool Helper library supports the following interrupts:

Name	Number	Meaning
INT_DIV0	0	Divide-error exception
INT_1	1	Debugger interrupt
INT_3	3	Breakpoint interrupt
INT_UDINSTR	6	Invalid-opcode exception
INT_STKFAULT	12	Stack exception
INT_GPFAULT	13	General protection violation
INT_BADPAGEFAULT	14	Page fault not caused by normal virtual-memory operation
INT_CTLALTSYS RQ	256	User pressed CTRL+ALT+SYS RQ

The Tool Helper library returns interrupt numbers as word values. Normal software interrupts and processor faults are represented by numbers in the range 0 through 255. Interrupts specific to Tool Helper are represented by numbers greater than 255.

Some developers may wish to use CTRL+ALT+SYS RQ (Interrupt 256) to break into the debugger. Be cautious about implementing this interrupt, because the point at which execution stops will probably be in a sensitive part of the Windows kernel. All **InterruptRegisterCallback** functions must be page-locked to prevent problems when this interrupt is used. In addition, the debugger probably will not be able to perform user-interface functions. However, the debugger can use Tool Helper functions to set breakpoints and gather information. The debugger may also be able to use a debugging terminal or secondary screen to display information.

Low-stack Faults

A low-stack fault occurs when inadequate stack space is available on the faulting application's stack. For example, if any fault occurs when there is less than 128 bytes of stack space available or if runaway recursion depletes the stack, a low-stack fault occurs. The Tool Helper library processes a low-stack fault differently than it processes other faults.

A low-stack fault is indicated by the high-order bit of the interrupt number being set. For example, if a stack fault occurs and the SP value becomes invalid, the Tool Helper library will return the fault number as 0x800C rather than 0x000C.

Interrupt handlers designed to process low-stack faults must be aware that the Tool Helper library has passed a fault frame on a stack other that the faulting application's stack. The SS:SP value is on the stack because it was pushed before the rest of the information in the stack frame. The SS:SP value is available only for advisory purposes.

An interrupt handler should never restart the faulting instruction, because this will cause the system to crash. The handler may terminate the application with **TerminateApp** or pass the fault to the next handler in the interrupt-handler list.

Interrupt handlers should not assume that all stack faults are low-stack faults. For example, if an application accesses a stack-relative variable that is out of range, a stack fault will occur. This type of fault can be processed in the same manner as any general protection (GP) fault. If the high-order bit of the interrupt number is not set, the instruction can be restarted.

Interrupt handlers also should not assume that all low-stack faults are stack faults. Any fault that occurs when there is less than 128 bytes of stack available will cause a low-stack fault.

Interrupt callback functions that are not designed to process low-stack faults should execute an **retf** instruction so that the Tool Helper library will pass the fault to the next appropriate handler in the interrupt-handler list.

See Also **Catch, InterruptUnRegister, NotifyRegister, NotifyUnRegister, TerminateApp, Throw**

InterruptUnRegister 3.1

Syntax #include <toolhelp.h>
BOOL InterruptUnRegister(htask)

function InterruptUnRegister(hTask: THandle): Bool;

The **InterruptUnRegister** function restores the default interrupt handle for system interrupts.

Parameters *htask* Identifies the task. If this value is NULL, it identifies the current task.

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

Comments After this function is executed, the Tool Helper library will pass all interrupts it receives to the system's default interrupt handler.

See Also **InterruptRegister, NotifyRegister, NotifyUnRegister, TerminateApp**

IsBadCodePtr 3.1

Syntax BOOL IsBadCodePtr(lpfn)

function IsBadCodePtr(lpfn: TFarProc): Bool;

The **IsBadCodePtr** function determines whether a pointer to executable code is valid.

Parameters *lpfn* Points to a function.

Return Value The return value is nonzero if the pointer is bad (that is, if it does not point to executable code). The return value is zero if the pointer is good.

See Also **IsBadHugeReadPtr, IsBadHugeWritePtr, IsBadReadPtr, IsBadStringPtr, IsBadWritePtr**

IsBadHugeReadPtr 3.1

Syntax BOOL IsBadHugeReadPtr(lp, cb)

function IsBadHugeReadPtr(lp: Pointer; cb: Longint): Bool;

The **IsBadHugeReadPtr** function determines whether a huge pointer to readable memory is valid.

Parameters

lp	Points to the beginning of a block of allocated memory. The data object may reside anywhere in memory and may exceed 64K in size.
cb	Specifies the number of bytes of memory that were allocated.

Return Value The return value is nonzero if the pointer is bad (that is, if it does not point to readable memory of the specified size). The return value is zero if the pointer is good.

See Also **IsBadCodePtr, IsBadHugeWritePtr, IsBadReadPtr, IsBadStringPtr, IsBadWritePtr**

IsBadHugeWritePtr 3.1

Syntax BOOL IsBadHugeWritePtr(lp, cb)

function IsBadHugeWritePtr(lp: Pointer; cb: Longint): Bool;

The **IsBadHugeWritePtr** function determines whether a huge pointer to writable memory is valid.

Parameters

lp	Points to the beginning of a block of allocated memory. The data object may reside anywhere in memory and may exceed 64K in size.
cb	Specifies the number of bytes of memory that were allocated.

Return Value The return value is nonzero if the pointer is bad (that is, if it does not point to writable memory of the specified size). The return value is zero if the pointer is good.

See Also **IsBadCodePtr, IsBadHugeReadPtr, IsBadReadPtr, IsBadStringPtr, IsBadWritePtr**

IsBadReadPtr 3.1

Syntax BOOL IsBadReadPtr(lp, cb)

function IsBadReadPtr(lp: Pointer; cb: Word): Bool;

The **IsBadReadPtr** function determines whether a pointer to readable memory is valid.

Parameters

lp	Points to the beginning of a block of allocated memory.
cb	Specifies the number of bytes of memory that were allocated.

Return Value The return value is nonzero if the pointer is bad (that is, if it does not point to readable memory of the specified size). The return value is zero if the pointer is good.

See Also **IsBadCodePtr**, **IsBadHugeReadPtr**, **IsBadHugeWritePtr**, **IsBadStringPtr**, **IsBadWritePtr**

IsBadStringPtr 3.1

Syntax BOOL IsBadStringPtr(lpsz, cchMax)

function IsBadStringPtr(lpsz: PChar; cchMax: Word): Bool;

The **IsBadStringPtr** function determines whether a pointer to a string is valid.

Parameters

lpsz	Points to a null-terminated string.
cchMax	Specifies the maximum size of the string, in bytes.

Return Value The return value is nonzero if the pointer is bad (that is, if it does not point to a string of the specified size). The return value is zero if the pointer is good.

See Also **IsBadCodePtr**, **IsBadHugeReadPtr**, **IsBadHugeWritePtr**, **IsBadReadPtr**, **IsBadWritePtr**

IsBadWritePtr 3.1

Syntax BOOL IsBadWritePtr(lp, cb)

function IsBadWritePtr(lp: Pointer; cb: Word): Bool;

The **IsBadWritePtr** function determines whether a pointer to writable memory is valid.

Parameters

lp	Points to the beginning of a block of allocated memory.
cb	Specifies the number of bytes of memory that were allocated.

Return Value The return value is nonzero if the pointer is bad (that is, if it does not point to writable memory of the specified size). The return value is zero if the pointer is good.

See Also **IsBadCodePtr, IsBadHugeReadPtr, IsBadHugeWritePtr, IsBadReadPtr, IsBadStringPtr**

IsGDIObject 3.1

Syntax BOOL IsGDIObject(hobj)

function IsGDIObject(Obj: THandle): Bool;

The **IsGDIObject** function determines whether the specified handle is not the handle of a graphics device interface (GDI) object.

Parameters

hobj	Specifies a handle to test.

Return Value The return value is nonzero if the handle may be the handle of a GDI object. It is zero if the handle is not the handle of a GDI object.

Comments An application cannot use **IsGDIObject** to guarantee that a given handle is to a GDI object. However, this function can be used to guarantee that a given handle is not to a GDI object.

See Also **GetObject**

IsMenu 3.1

Syntax BOOL IsMenu(hmenu)

function IsMenu(Menu: HMenu): Bool;

The **IsMenu** function determines whether the given handle is a menu handle.

Parameters *hmenu* Identifies the handle to be tested.

Return Value The return value is zero if the handle is definitely *not* a menu handle. A nonzero return value does not guarantee that the handle is a menu handle, however; for nonzero return values, the application should conduct further tests to verify the handle.

Comments An application should use this function only to ensure that a given handle is *not* a menu handle.

See Also **CreateMenu, CreatePopupMenu, DestroyMenu, GetMenu**

IsTask 3.1

Syntax BOOL IsTask(htask)

function IsTask(Task: THandle): Bool;

The **IsTask** function determines whether the given task handle is valid.

Parameters *htask* Identifies a task.

Return Value The return value is nonzero if the task handle is valid. Otherwise, it is zero.

JournalPlaybackProc 3.1

Syntax LRESULT CALLBACK JournalPlaybackProc(code, wParam, lParam)

The **JournalPlaybackProc** function is a library-defined callback function that a library can use to insert mouse and keyboard messages into the system message queue. Typically, a library uses this function to play back a series of mouse and keyboard messages that were recorded earlier by using the **JournalRecordProc** function. Regular mouse and keyboard input is disabled as long as a **JournalPlaybackProc** function is installed.

Parameters

code
: Specifies whether the callback function should process the message or call the **CallNextHookEx** function. If this parameter is less than zero, the callback function should pass the message to **CallNextHookEx** without further processing.

wParam
: Specifies a NULL value.

lParam
: Points to an **EVENTMSG** structure that represents the message being processed by the callback function. The **EVENTMSG** structure has the following form:

```
typedef struct tagEVENTMSG {     /* em */
    UINT  message;
    UINT  paramL;
    UINT  paramH;
    DWORD time;
} EVENTMSG;
```

Return Value The callback function should return a value that represents the amount of time, in clock ticks, that the system should wait before processing the message. This value can be computed by calculating the difference between the **time** members of the current and previous input messages. If the function returns zero, the message is processed immediately.

Comments The **JournalPlaybackProc** function should copy an input message to the *lParam* parameter. The message must have been recorded by using a **JournalRecordProc** callback function, which should not modify the message.

Once the function returns control to the system, the message continues to be processed. If the *code* parameter is HC_SKIP, the filter function should prepare to return the next recorded event message on its next call.

This callback function should reside in a dynamic-link library.

An application must install the callback function by specifying the WH_JOURNALPLAYBACK filter type and the procedure-instance address of the callback function in a call to the **SetWindowsHookEx** function.

JournalPlaybackProc is a placeholder for the library-defined function name. The actual name must be exported by including it in an **EXPORTS** statement in the library's module-definition file.

See Also **CallNextHookEx, JournalRecordProc, SetWindowsHookEx**

JournalRecordProc 3.1

Syntax LRESULT CALLBACK JournalRecordProc(code, wParam, lParam)

The **JournalRecordProc** function is a library-defined callback function that records messages that the system removes from the system message queue. Later, a library can use a **JournalPlaybackProc** function to play back the messages.

Parameters

code Specifies whether the callback function should process the message or call the **CallNextHookEx** function. If this parameter is less than zero, the callback function should pass the message to **CallNextHookEx** without further processing.

wParam Specifies a NULL value.

lParam Points to an **MSG** structure. The **MSG** structure has the following form:

```
typedef struct tagMSG {     /* msg */
    HWND   hwnd;
    UINT   message;
    WPARAM wParam;
    LPARAM lParam;
    DWORD  time;
    POINT  pt;
} MSG;
```

Return Value The callback function should return zero.

Comments A **JournalRecordProc** callback function should copy but not modify the messages. After control returns to the system, the message continues to be processed. The callback function does not require a return value.

This callback function must be in a dynamic-link library.

An application must install the callback function by specifying the WH_JOURNALRECORD filter type and the procedure-instance address of the callback function in a call to the **SetWindowsHookEx** function.

JournalRecordProc is a placeholder for the library-defined function name. The actual name must be exported by including it in an **EXPORTS** statement in the library's module-definition file.

See Also **CallNextHookEx, JournalPlaybackProc, SetWindowsHookEx**

KeyboardProc 3.1

Syntax LRESULT CALLBACK KeyboardProc(code, wParam, lParam)

The **KeyboardProc** function is a library-defined callback function that the system calls whenever the application calls the **GetMessage** or **PeekMessage** function and there is a WM_KEYUP or WM_KEYDOWN keyboard message to process.

Parameters

code Specifies whether the callback function should process the message or call the **CallNextHookEx** function. If this value is HC_NOREMOVE, the application is using the **PeekMessage** function with the PM_NOREMOVE option, and the message will not be removed from the system queue. If this value is less than zero, the callback function should pass the message to **CallNextHookEx** without further processing.

wParam Specifies the virtual-key code of the given key.

lParam Specifies the repeat count, scan code, extended key, previous key state, context code, and key-transition state, as shown in the following table. (Bit 0 is the low-order bit):

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 The callback function should return 0 if the message should be processed by the system; it should return 1 if the message should be discarded.

Comments This callback function must be in a dynamic-link library.

An application must install the callback function by specifying the WH_KEYBOARD filter type and the procedure-instance address of the callback function in a call to the **SetWindowsHookEx** function.

KeyboardProc is a placeholder for the library-defined function name. The actual name must be exported by including it in an **EXPORTS** statement in the library's module-definition file.

See Also **CallNextHookEx, GetMessage, PeekMessage, SetWindowsHookEx**

LibMain 2.x

Syntax int CALLBACK LibMain(hinst, wDataSeg, cbHeapSize, lpszCmdLine)

The **LibMain** function is called by the system to initialize a dynamic-link library (DLL). A DLL must contain the **LibMain** function if the library is linked with the file LIBENTRY.OBJ.

Parameters

hinst	Identifies the instance of the DLL.
wDataSeg	Specifies the value of the data segment (DS) register.
cbHeapSize	Specifies the size of the heap defined in the module-definition file. (The LibEntry routine in LIBENTRY.OBJ uses this value to initialize the local heap.)
lpszCmdLine	Points to a null-terminated string specifying command-line information. This parameter is rarely used by DLLs.

Return Value The function should return 1 if it is successful. Otherwise, it should return 0.

Comments The **LibMain** function is called by LibEntry, which is called by Windows when the DLL is loaded. The LibEntry routine is provided in the LIBENTRY.OBJ module. LibEntry initializes the DLL's heap (if a **HEAPSIZE** value is specified in the DLL's module-definition file) before calling the **LibMain** function.

Example The following example shows a typical **LibMain** function:

```
int CALLBACK LibMain(HINSTANCE hinst, WORD wDataSeg, WORD cbHeap,
    LPSTR lpszCmdLine )
{
    HGLOBAL    hgblClassStruct;
    LPWNDCLASS lpClassStruct;
    static HINSTANCE hinstLib;

    /* Has the library been initialized yet? */

    if (hinstLib == NULL) {
        hgblClassStruct = GlobalAlloc(GHND, sizeof(WNDCLASS));
        if (hgblClassStruct != NULL) {
            lpClassStruct = (LPWNDCLASS) GlobalLock(hgblClassStruct);
            if (lpClassStruct != NULL) {

                /* Define the class attributes. */

                lpClassStruct->style = CS_HREDRAW | CS_VREDRAW |
                    CS_DBLCLKS | CS_GLOBALCLASS;
                lpClassStruct->lpfnWndProc = DllWndProc;
                lpClassStruct->cbWndExtra = 0;
                lpClassStruct->hInstance = hinst;
                lpClassStruct->hIcon = NULL;
                lpClassStruct->hCursor = LoadCursor(NULL, IDC_ARROW);
                lpClassStruct->hbrBackground =
                    (HBRUSH) (COLOR_WINDOW + 1);
                lpClassStruct->lpszMenuName = NULL;
                lpClassStruct->lpszClassName = "MyClassName";

                hinstLib = (RegisterClass(lpClassStruct)) ?
                    hinst : NULL;

                GlobalUnlock(hgblClassStruct);
            }

            GlobalFree(hgblClassStruct);
        }
    }
    return (hinstLib ? 1 : 0);  /* return 1 = success; 0 = fail */
}
```

See Also **GlobalAlloc, GlobalFree, GlobalLock, GlobalUnlock, WEP**

LineDDAProc 3.1

Syntax void CALLBACK LineDDAProc(xPos, yPos, lpData)

The **LineDDAProc** function is an application-defined callback function that processes coordinates from the **LineDDA** function.

Parameters

xPos	Specifies the x-coordinate of the current point.
yPos	Specifies the y-coordinate of the current point.
lpData	Points to the application-defined data.

Return Value This function does not return a value.

Comments An application must register this function by passing its address to the **LineDDA** function.

LineDDAProc is a placeholder for the application-defined function name. The actual name must be exported by including it in an **EXPORTS** statement in the application's module-definition file.

See Also **LineDDA**

LoadProc 2.x

Syntax HGLOBAL CALLBACK LoadProc(hglbMem, hinst, hrsrcResInfo)

The **LoadProc** function is an application-defined callback function that receives information about a resource to be locked and can process that information as needed.

Parameters

hglbMem	Identifies a memory object that contains a resource. This parameter is NULL if the resource has not yet been loaded.
hinst	Identifies the instance of the module whose executable file contains the resource.
hrsrcResInfo	Identifies the resource. The resource must have been created by using the **FindResource** function.

Return Value The return value is a global memory handle for memory that was allocated using the GMEM_DDESHARE flag in the **GlobalAlloc** function.

Comments If an attempt to lock the memory object identified by the *hglbMem* parameter fails, this means the resource has been discarded and must be reloaded.

LoadProc is a placeholder for the application-defined function name. The actual name must be exported by including it in an **EXPORTS** statement in the application's module-definition file.

See Also **FindResource, GlobalAlloc, SetResourceHandler**

LocalFirst 3.1

Syntax #include <toolhelp.h>
BOOL LocalFirst(lple, hglbHeap)

function LocalFirst(lpLocal: PLocalEntry; hHeap: THandle): Bool;

The **LocalFirst** function fills the specified structure with information that describes the first object on the local heap.

Parameters

lple Points to a **LOCALENTRY** structure that will receive information about the local memory object. The **LOCALENTRY** structure has the following form:

```
#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;
```

hglbHeap Identifies the local heap.

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

Comments The **LocalFirst** function can be used to begin a local heap walk. An application can examine subsequent objects on the local heap by using the **LocalNext** function.

Before calling **LocalFirst**, an application must initialize the **LOCALENTRY** structure and specify its size, in bytes, in the **dwSize** member.

See Also **LocalInfo, LocalNext**

LocalInfo 3.1

Syntax #include <toolhelp.h>
BOOL LocalInfo(lpli, hglbHeap)

function LocalInfo(lpLocal: PLocalInfo; hHeap: THandle): Bool;

The **LocalInfo** function fills the specified structure with information that describes the local heap.

Parameters

lpli Points to a **LOCALINFO** structure that will receive information about the local heap. The **LOCALINFO** structure has the following form:

```
#include <toolhelp.h>

typedef struct tagLOCALINFO {  /* li */
    DWORD  dwSize;
    WORD   wcItems;
} LOCALINFO;
```

hglbHeap Identifies the local heap to be described.

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

Comments The information in the **LOCALINFO** structure can be used to determine how much memory to allocate for a local heap walk.

Before calling **LocalInfo**, an application must initialize the **LOCALINFO** structure and specify its size, in bytes, in the **dwSize** member.

See Also **LocalFirst, LocalNext**

LocalNext 3.1

Syntax #include <toolhelp.h>
BOOL LocalNext(lple)

function LocalNext(lpLocal: PLocalEntry): Boolean;

The **LocalNext** function fills the specified structure with information that describes the next object on the local heap.

Parameters *lple* Points to a **LOCALENTRY** structure that will receive information about the local memory object. The **LOCALENTRY** structure has the following form:

```
#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;
```

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

Comments The **LocalNext** function can be used to continue a local heap walk started by the **LocalFirst** function.

See Also **LocalFirst, LocalInfo**

LockInput 3.1

Syntax BOOL LockInput(hReserved, hwndInput, fLock)

function LockInput(h1: THandle; hwndInput: HWnd; fLock: Bool): Bool;

The **LockInput** function locks input to all tasks except the current one, if the *fLock* parameter is TRUE. The given window is made system modal; that is, it will receive all input. If *fLock* is FALSE, **LockInput** unlocks input and restores the system to its unlocked state.

Parameters

hReserved	This parameter is reserved and must be NULL.
hwndInput	Identifies the window that is to receive all input. This window must be in the current task. If *fLock* is FALSE, this parameter should be NULL.
fLock	Indicates whether to lock or unlock input. A value of TRUE locks input; a value of FALSE unlocks input.

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

Comments Before entering hard mode, a Windows-based debugger calls **LockInput**, specifying TRUE for the *fLock* parameter. This action saves the current global state. To exit hard mode, the debugger calls **LockInput**, specifying FALSE for *fLock*. This restores the global state to the conditions that existed when the debugger entered hard mode. A debugger must restore the global state before exiting. Calls to **LockInput** cannot be nested.

See Also **DirectedYield**

LockWindowUpdate 3.1

Syntax BOOL LockWindowUpdate(hwndLock)

function LockWindowUpdate(Wnd: HWnd): Bool;

The **LockWindowUpdate** function disables or reenables drawing in the given window. A locked window cannot be moved. Only one window can be locked at a time.

Parameters

hwndLock	Identifies the window in which drawing will be disabled. If this parameter is NULL, drawing in the locked window is enabled.

Return Value The return value is nonzero if the function is successful. It is zero if a failure occurs or if the **LockWindowUpdate** function has been used to lock another window.

Comments If an application with a locked window (or any locked child windows) calls the **GetDC**, **GetDCEx**, or **BeginPaint** function, the called function returns a device context whose visible region is empty. This will occur until the application unlocks the window by calling **LockWindowUpdate**, specifying a value of NULL for *hwndLock*.

While window updates are locked, the system keeps track of the bounding rectangle of any drawing operations to device contexts associated with a locked window. When drawing is reenabled, this bounding rectangle is invalidated in the locked window and its child windows to force an eventual WM_PAINT message to update the screen. If no drawing has occurred while the window updates were locked, no area is invalidated.

The **LockWindowUpdate** function does not make the given window invisible and does not clear the WS_VISIBLE style bit.

LogError 3.1

Syntax void LogError(uErr, lpvInfo)

procedure LogError(Err: Word; Info: Pointer);

The **LogError** function identifies the most recent system error. An application's interrupt callback function typically calls **LogError** to return error information to the user.

Parameters *uErr* Specifies the type of error that occurred. The *lpvInfo* parameter may point to more information about the error, depending on the value of *uErr*. This parameter may be one or more of the following values:

Value	Meaning
ERR_ALLOCRES	**AllocResource** failed.
ERR_BADINDEX	Bad index to **GetClassLong**, **GetClassWord**, **GetWindowLong**, **GetWindowWord**, **SetClassLong**, **SetClassWord**, **SetWindowLong**, or **SetWindowWord**.
ERR_BYTE	Invalid 8-bit parameter.
ERR_CREATEDC	**CreateCompatibleDC**, **CreateDC**, or **CreateIC** failed.
ERR_CREATEDLG	Could not create dialog box because **LoadMenu** failed.
ERR_CREATEDLG2	Could not create dialog box because **CreateWindow** failed.
ERR_CREATEMENU	Could not create menu.
ERR_CREATEMETA	**CreateMetaFile** failed.
ERR_CREATEWND	Could not create window because the class was not found.
ERR_DCBUSY	Device context (DC) cache is full.
ERR_DELOBJSELECTED	Program is trying to delete a bitmap that is selected into the DC.
ERR_DWORD	Invalid 32-bit parameter.
ERR_GALLOC	**GlobalAlloc** failed.
ERR_GLOCK	**GlobalLock** failed.
ERR_GREALLOC	**GlobalReAlloc** failed.
ERR_LALLOC	**LocalAlloc** failed.
ERR_LLOCK	**LocalLock** failed.
ERR_LOADMENU	**LoadMenu** failed.

Value	Meaning
ERR_LOADMODULE	**LoadModule** failed.
ERR_LOADSTR	**LoadString** failed.
ERR_LOCKRES	**LockResource** failed.
ERR_LREALLOC	**LocalReAlloc** failed.
ERR_NESTEDBEGINPAINT	Program contains nested **BeginPaint** calls.
ERR_REGISTERCLASS	**RegisterClass** failed because the class is already registered.
ERR_SELBITMAP	Program is trying to select a bitmap that is already selected.
ERR_SIZE_MASK	Identifies which 2 bits of *uErr* specify the size of the invalid parameter.
ERR_STRUCEXTRA	Program is using unallocated space.
ERR_WARNING	A non-fatal error occurred.
ERR_WORD	Invalid 16-bit parameter.

lpvInfo Points to more information about the error. The value of *lpvInfo* depends on the value of *uErr*. If the value of (*uErr* & ERR_SIZE_MASK) is 0, *lpvInfo* is undefined. Currently, no *uErr* code has defined meanings for *lpvInfo*.

Return Value This function does not return a value.

Comments The errors identified by **LogError** may be trapped by the callback function that **NotifyRegister** installs.

Error values whose low 12 bits are less than 0x07FF are reserved for use by Windows.

See Also **LogParamError, NotifyRegister**

LogParamError 3.1

Syntax void LogParamError(uErr, lpfn, lpvParam)

procedure LogParamError(Err: Word; fn: TFarProc; Param: Pointer);

The **LogParamError** function identifies the most recent parameter validation error. An application's interrupt callback function typically calls **LogParamError** to return information about an invalid parameter to the user.

Parameters *uErr*

Specifies the type of parameter validation error that occurred. The *lpvParam* parameter may point to more information about the error, depending on the value of *uErr*. This parameter may be one or more of the following values:

Value	Meaning
ERR_BAD_ATOM	Invalid atom.
ERR_BAD_CID	Invalid communications identifier (CID).
ERR_BAD_COORDS	Invalid x,y coordinates.
ERR_BAD_DFLAGS	Invalid 32-bit flags.
ERR_BAD_DINDEX	Invalid 32-bit index or index out-of-range.
ERR_BAD_DVALUE	Invalid 32-bit signed or unsigned value.
ERR_BAD_FLAGS	Invalid bit flags.
ERR_BAD_FUNC_PTR	Invalid function pointer.
ERR_BAD_GDI_OBJECT	Invalid graphics device interface (GDI) object.
ERR_BAD_GLOBAL_HANDLE	Invalid global handle.
ERR_BAD_HANDLE	Invalid generic handle.
ERR_BAD_HBITMAP	Invalid bitmap handle.
ERR_BAD_HBRUSH	Invalid brush handle.
ERR_BAD_HCURSOR	Invalid cursor handle.
ERR_BAD_HDC	Invalid device context (DC) handle.
ERR_BAD_HDRVR	Invalid driver handle.
ERR_BAD_HDWP	Invalid handle of a window-position structure.
ERR_BAD_HFILE	Invalid file handle.
ERR_BAD_HFONT	Invalid font handle.
ERR_BAD_HICON	Invalid icon handle.
ERR_BAD_HINSTANCE	Invalid instance handle.
ERR_BAD_HMENU	Invalid menu handle.
ERR_BAD_HMETAFILE	Invalid metafile handle.
ERR_BAD_HMODULE	Invalid module handle.
ERR_BAD_HPALETTE	Invalid palette handle.
ERR_BAD_HPEN	Invalid pen handle.
ERR_BAD_HRGN	Invalid region handle.
ERR_BAD_HWND	Invalid window handle.
ERR_BAD_INDEX	Invalid index or index out-of-range.
ERR_BAD_LOCAL_HANDLE	Invalid local handle.

Value	Meaning
ERR_BAD_PTR	Invalid pointer.
ERR_BAD_SELECTOR	Invalid selector.
ERR_BAD_STRING_PTR	Invalid zero-terminated string pointer.
ERR_BAD_VALUE	Invalid 16-bit signed or unsigned value.
ERR_BYTE	Invalid 8-bit parameter.
ERR_DWORD	Invalid 32-bit parameter.
ERR_PARAM	A parameter validation error occurred. This flag is always set.
ERR_SIZE_MASK	Identifies which 2 bits of *uErr* specify the size of the invalid parameter.
ERR_WARNING	An invalid parameter was detected, but the error is not serious enough to cause the function to fail. The invalid parameter is reported, but the call runs as usual.
ERR_WORD	Invalid 16-bit parameter.

lpfn Specifies the address at which the parameter error occurred. This value is NULL if the address is unknown.

lpvParam Points to more information about the error. The value of *lpvParam* depends on the value of *uErr*. If the value of (*uErr* & ERR_SIZE_MASK) is 0, *lpvParam* is undefined. Currently, no *uErr* code has defined meanings for *lpvParam*.

Return Value This function does not return a value.

Comments The errors identified by **LogParamError** may be trapped by the callback function that **NotifyRegister** installs.

Error values whose low 12 bits are less than 0x07FF are reserved for use by Windows.

The size of the value passed in *lpvParam* is determined by the values of the bits selected by ERR_SIZE_MASK, as follows:

```
switch (err & ERR_SIZE_MASK)
{
case ERR_BYTE:          /* 8-bit invalid parameter */
    b = LOBYTE(param);
    break;
```

```
    case ERR_WORD:          /* 16-bit invalid parameter */
        w = LOWORD(param);
        break;

    case ERR_DWORD:         /* 32-bit invalid parameter */
        l = (DWORD)param;
        break:

    default:                /* invalid parameter value is unknown */
        break;
}
```

See Also **LogError, NotifyRegister**

LZClose 3.1

Syntax #include <lzexpand.h>
void LZClose(hf)

procedure LZClose(LZFile: Integer);

The **LZClose** function closes a file that was opened by the **LZOpenFile** or **OpenFile** function.

Parameters *hf* Identifies the source file.

Return Value This function does not return a value.

Comments If the file was compressed by Microsoft File Compression Utility (COMPRESS.EXE) and opened by the **LZOpenFile** function, **LZClose** frees any global heap space that was required to expand the file.

Example The following example uses **LZClose** to close a file opened by **LZOpenFile**:

```
char szSrc[] = {"readme.txt"};
char szDst[] = {"readme.bak"};
OFSTRUCT ofStrSrc;
OFSTRUCT ofStrDest;
HFILE hfSrcFile, hfDstFile;

/* Open the source file. */

hfSrcFile = LZOpenFile(szSrc, &ofStrSrc, OF_READ);

/* Create the destination file. */

hfDstFile = LZOpenFile(szDst, &ofStrDest, OF_CREATE);

/* Copy the source file to the destination file. */

LZCopy(hfSrcFile, hfDstFile);
```

```
/* Close the files. */

LZClose(hfSrcFile);
LZClose(hfDstFile);
```

See Also **OpenFile, LZOpenFile**

LZCopy 3.1

Syntax #include <lzexpand.h>
LONG LZCopy(hfSource, hfDest)

function LZCopy(Source, Dest: Integer): Longint;

The **LZCopy** function copies a source file to a destination file. If the source file was compressed by Microsoft File Compression Utility (COMPRESS.EXE), this function creates a decompressed destination file. If the source file was not compressed, this function duplicates the original file.

Parameters

hfSource Identifies the source file. (This handle is returned by the **LZOpenFile** function when a compressed file is opened.)

hfDest Identifies the destination file.

Return Value The return value is the size, in bytes, of the destination file if the function is successful. Otherwise, it is an error value that is less than zero and may be one of the following:

Value	Meaning
LZERROR_BADINHANDLE	The handle identifying the source file was not valid.
LZERROR_BADOUTHANDLE	The handle identifying the destination file was not valid.
LZERROR_GLOBALLOC	There is insufficient memory for the required buffers.
LZERROR_GLOBLOCK	The handle identifying the internal data structures is invalid.
LZERROR_READ	The source file format was not valid.
LZERROR_UNKNOWNALG	The source file was compressed with an unrecognized compression algorithm.
LZERROR_WRITE	There is insufficient space for the output file.

Comments This function is designed for single-file copy operations. (Use the **CopyLZFile** function for multiple-file copy operations.)

If the function is successful, the file identified by *hfDest* is uncompressed.

If the source or destination file is opened by a C run-time function (rather than the **_lopen** or **OpenFile** function), it must be opened in binary mode.

Example The following example uses the **LZCopy** function to copy a file:

```
char szSrc[] = {"readme.txt"};
char szDst[] = {"readme.bak"};
OFSTRUCT ofStrSrc;
OFSTRUCT ofStrDest;
HFILE hfSrcFile, hfDstFile;

/* Open the source file. */

hfSrcFile = LZOpenFile(szSrc, &ofStrSrc, OF_READ);

/* Create the destination file. */

hfDstFile = LZOpenFile(szDst, &ofStrDest, OF_CREATE);

/* Copy the source file to the destination file. */

LZCopy(hfSrcFile, hfDstFile);

/* Close the files. */

LZClose(hfSrcFile);
LZClose(hfDstFile);
```

See Also **CopyLZFile, _lopen, LZOpenFile, OpenFile**

LZDone 3.1

Syntax #include <lzexpand.h>
void LZDone(void)

procedure LZDone;

The **LZDone** function frees buffers that the **LZStart** function allocated for multiple-file copy operations.

Parameters This function has no parameters.

Return Value This function does not return a value.

Comments Applications that copy multiple files should call **LZStart** before copying the files with the **CopyLZFile** function. **LZStart** allocates buffers for the file copy operations.

Example The following example uses **LZDone** to free buffers allocated by **LZStart:**

```
#define NUM_FILES   4

char *szSrc[NUM_FILES] =
    {"readme.txt", "data.txt", "update.txt", "list.txt"};
char*szDest[NUM_FILES]=
    {"readme.bak", "data.bak", "update.bak", "list.bak"};
OFSTRUCT ofStrSrc;
OFSTRUCT ofStrDest;
HFILE hfSrcFile, hfDstFile;
int i;

/* Allocate internal buffers for the CopyLZFile function. */

LZStart();

/* Open, copy, and then close the files. */

for (i = 0; i < NUM_FILES; i++) {
    hfSrcFile = LZOpenFile(szSrc[i], &ofStrSrc, OF_READ);
    hfDstFile = LZOpenFile(szDest[i], &ofStrDest, OF_CREATE);
    CopyLZFile(hfSrcFile, hfDstFile);
    LZClose(hfSrcFile);
    LZClose(hfDstFile);
}

LZDone(); /* free the internal buffers */
```

See Also **CopyLZFile, LZCopy, LZStart**

LZInit 3.1

Syntax #include <lzexpand.h>
HFILE LZInit(hfSrc)

function LZInit(Source: Integer): Integer;

The **LZInit** function allocates memory for, creates, and initializes the internal data structures that are required to decompress files.

Parameters *hfSrc* Identifies the source file.

Return Value The return value is the original file handle if the function is successful and the file is not compressed. If the function is successful and the file is compressed, the return value is a new file handle. If the function fails, the return value is an error value that is less than zero and may be one of the following:

Value	Meaning
LZERROR_BADINHANDLE	The handle identifying the source file is invalid.
LZERROR_GLOBALLOC	There is insufficient memory for the required internal data structures. This value is returned when an application attempts to open more than 16 files.
LZERROR_GLOBLOCK	The handle identifying global memory is invalid. (The internal call to the **GlobalLock** function failed.)
LZERROR_READ	The source file format is invalid.
LZERROR_UNKNOWNALG	The file was compressed with an unrecognized compression algorithm.

Comments A maximum of 16 compressed files can be open at any given time.

Example The following example uses **LZInit** to initialize the internal structures that are required to decompress a file:

```
char szSrc[] = {"readme.cmp"};
char szFileName[128];
OFSTRUCT ofStrSrc;
OFSTRUCT ofStrDest;
HFILE hfSrcFile, hfDstFile, hfCompFile;
int cbRead;
BYTE abBuf[512];

/* Open the compressed source file. */

hfSrcFile = OpenFile(szSrc, &ofStrSrc, OF_READ);

/*
 * Initialize internal data structures for the decompression
 * operation.
 */

hfCompFile = LZInit(hfSrcFile);

/* Retrieve the original name for the compressed file. */

GetExpandedName(szSrc, szFileName);

/* Create the destination file using the original name. */

hfDstFile = LZOpenFile(szFileName, &ofStrDest, OF_CREATE);

/* Copy the compressed source file to the destination file. */

do {
    if ((cbRead = LZRead(hfCompFile, abBuf, sizeof(abBuf))) > 0)
        _lwrite(hfDstFile, abBuf, cbRead);
```

```
        else {
            .
            . /* handle error condition */
            .
        }
    } while (cbRead == sizeof(abBuf));

    /* Close the files. */

    LZClose(hfSrcFile);
    LZClose(hfDstFile);
```

LZOpenFile 3.1

Syntax #include <lzexpand.h>
HFILE LZOpenFile(lpszFile, lpof, style)

function LZOpenFile(FileName: PChar; var ReOpenBuf: TOFStruct; Style: Word): Integer;

The **LZOpenFile** function creates, opens, reopens, or deletes the file specified by the string to which *lpszFile* points.

Parameters

lpszFile Points to a string that specifies the name of a file.

lpof Points to the **OFSTRUCT** structure that is to receive information about the file when the file is opened. The structure can be used in subsequent calls to **LZOpenFile** to refer to the open file.

The **szPathName** member of this structure contains characters from the OEM character set.

style Specifies the action to be taken. These styles can be combined by using the bitwise OR operator:

Value	Meaning
OF_CANCEL	Adds a Cancel button to the OF_PROMPT dialog box. Choosing the Cancel button directs **LZOpenFile** to return a file-not-found error message.
OF_CREATE	Directs **LZOpenFile** to create a new file. If the file already exists, it is truncated to zero length.
OF_DELETE	Deletes the file.
OF_EXIST	Opens the file, and then closes it. This action is used to test for file existence.
OF_PARSE	Fills the **OFSTRUCT** structure, but carries out no other action.

Value	Meaning
OF_PROMPT	Displays a dialog box if the requested file does not exist. The dialog box informs the user that Windows cannot find the file and prompts the user to insert the disk containing the file in drive A.
OF_READ	Opens the file for reading only.
OF_READWRITE	Opens the file for reading and writing.
OF_REOPEN	Opens the file using information in the reopen buffer.
OF_SHARE_DENY_NONE	Opens the file without denying other programs read access or write access to the file. **LZOpenFile** fails if the file has been opened in compatibility mode by any other program.
OF_SHARE_DENY_READ	Opens the file and denies other programs read access to the file. **LZOpenFile** fails if the file has been opened in compatibility mode or for read access by any other program.
OF_SHARE_DENY_WRITE	Opens the file and denies other programs write access to the file. **LZOpenFile** fails if the file has been opened in compatibility mode or for write access by any other program.
OF_SHARE_EXCLUSIVE	Opens the file in exclusive mode, denying other programs both read access and write access to the file. **LZOpenFile** fails if the file has been opened in any other mode for read access or write access, even by the current program.
OF_WRITE	Opens the file for writing only.

Return Value

The return value is a handle identifying the file if the function is successful and the value specified by *style* is not OF_READ. If the file is compressed and opened with *style* set to the OF_READ value, the return value is a special file handle. If the function fails, the return value is –1.

Comments

If *style* is OF_READ (or OF_READ and any of the OF_SHARE_ flags) and the file is compressed, **LZOpenFile** calls the **LZInit** function, which performs the required initialization for the decompression operations.

Example

The following example uses **LZOpenFile** to open a source file and create a destination file into which the source file can be copied:

```
char szSrc[] = {"readme.txt"};
char szDst[] = {"readme.bak"};
OFSTRUCT ofStrSrc;
OFSTRUCT ofStrDest;
HFILE hfSrcFile, hfDstFile;
```

```
/* Open the source file. */

hfSrcFile = LZOpenFile(szSrc, &ofStrSrc, OF_READ);

/* Create the destination file. */

hfDstFile = LZOpenFile(szDst, &ofStrDest, OF_CREATE);

/* Copy the source file to the destination file. */

LZCopy(hfSrcFile, hfDstFile);

/* Close the files. */

LZClose(hfSrcFile);
LZClose(hfDstFile);
```

See Also **LZInit**

LZRead 3.1

Syntax #include <lzexpand.h>
int LZRead(hf, lpvBuf, cb)

function LZRead(LZFile: Integer; Buf: PChar; Count: Integer): Integer;

The **LZRead** function reads into a buffer bytes from a file.

Parameters

hf	Identifies the source file.
lpvBuf	Points to a buffer that is to receive the bytes read from the file.
cb	Specifies the maximum number of bytes to be read.

Return Value The return value is the actual number of bytes read if the function is successful. Otherwise, it is an error value that is less than zero and may be any of the following:

Value	Meaning
LZERROR_BADINHANDLE	The handle identifying the source file was invalid.
LZERROR_BADVALUE	The *cb* parameter specified a negative value.
LZERROR_GLOBLOCK	The handle identifying required initialization data is invalid.
LZERROR_READ	The format of the source file was invalid.
LZERROR_UNKNOWNALG	The file was compressed with an unrecognized compression algorithm.

Comments If the file is not compressed, **LZRead** calls the **_lread** function, which performs the read operation.

If the file is compressed, **LZRead** emulates **_lread** on an expanded image of the file and copies the bytes of data into the buffer to which *lpvBuf* points.

If the source file was compressed by Microsoft File Compression Utility (COMPRESS.EXE), the **LZOpenFile**, **LZSeek**, and **LZRead** functions can be called instead of the **OpenFile**, **_llseek**, and **_lread** functions.

Example The following example uses **LZRead** to copy and decompress a compressed file:

```
char szSrc[] = {"readme.cmp"};
char szFileName[128];
OFSTRUCT ofStrSrc;
OFSTRUCT ofStrDest;
HFILE hfSrcFile, hfDstFile, hfCompFile;
int cbRead;
BYTE abBuf[512];

/* Open the compressed source file. */

hfSrcFile = OpenFile(szSrc, &ofStrSrc, OF_READ);

/*
 * Initialize internal data structures for the decompression
 * operation.
 */

hfCompFile = LZInit(hfSrcFile);

/* Retrieve the original name for the compressed file. */

GetExpandedName(szSrc, szFileName);

/* Create the destination file using the original name. */

hfDstFile = LZOpenFile(szFileName, &ofStrDest, OF_CREATE);

/* Copy the compressed source file to the destination file. */

do {
    if ((cbRead = LZRead(hfCompFile, abBuf, sizeof(abBuf))) > 0)
        _lwrite(hfDstFile, abBuf, cbRead);
    else {
        .
        . /* handle error condition */
        .
    }
} while (cbRead == sizeof(abBuf));

/* Close the files. */
```

```
LZClose(hfSrcFile);
LZClose(hfDstFile);
```

See Also **_llseek, _lread, LZOpenFile, LZRead, LZSeek**

LZSeek 3.1

Syntax #include <lzexpand.h>
LONG LZSeek(hf, lOffset, nOrigin)

function LZSeek(LZFile: Integer; SeekTo: Longint; Mode: Integer): Longint;

The **LZSeek** function moves a file pointer from its original position to a new position.

Parameters

hf Identifies the source file.

lOffset Specifies the number of bytes by which the file pointer should be moved.

nOrigin Specifies the starting position of the pointer. This parameter must be one of the following values:

Value	Meaning
0	Move the file pointer *lOffset* bytes from the beginning of the file.
1	Move the file pointer *lOffset* bytes from the current position.
2	Move the file pointer *lOffset* bytes from the end of the file.

Return Value The return value is the offset from the beginning of the file to the new pointer position, if the function is successful. Otherwise, it is an error value that is less than zero and may be one of the following:

Value	Meaning
LZERROR_BADINHANDLE	The handle identifying the source file was invalid.
LZERROR_BADVALUE	One of the parameters exceeds the range of valid values.
LZERROR_GLOBLOCK	The handle identifying the initialization data is invalid.

Comments If the file is not compressed, **LZSeek** calls the **_llseek** function and moves the file pointer by the specified offset.

If the file is compressed, **LZSeek** emulates **_llseek** on an expanded image of the file.

See Also **_llseek**

LZStart 3.1

Syntax #include <lzexpand.h>
int LZStart(void)

function LZStart: Integer;

The **LZStart** function allocates the buffers that the **CopyLZFile** function uses to copy a source file to a destination file.

Parameters This function has no parameters.

Return Value The return value is nonzero if the function is successful. Otherwise, it is LZERROR_GLOBALLOC.

Comments Applications that copy (or copy and decompress) multiple consecutive files should call the **LZStart**, **CopyLZFile**, and **LZDone** functions. Applications that copy a single file should call the **LZCopy** function.

Example The following example uses **LZStart** to allocate buffers used by **CopyLZFile**:

```
#define NUM_FILES   4

char *szSrc[NUM_FILES] =
    {"readme.txt", "data.txt", "update.txt", "list.txt"};
char*szDest[NUM_FILES]=
    {"readme.bak", "data.bak", "update.bak", "list.bak"};
OFSTRUCT ofStrSrc;
OFSTRUCT ofStrDest;
HFILE hfSrcFile, hfDstFile;
int i;

/* Allocate internal buffers for the CopyLZFile function. */

LZStart();

/* Open, copy, and then close the files. */

for (i = 0; i < NUM_FILES; i++) {
    hfSrcFile = LZOpenFile(szSrc[i], &ofStrSrc, OF_READ);
```

```
        hfDstFile = LZOpenFile(szDest[i], &ofStrDest, OF_CREATE);
        CopyLZFile(hfSrcFile, hfDstFile);
        LZClose(hfSrcFile);
        LZClose(hfDstFile);
    }

    LZDone(); /* free the internal buffers */
```

See Also **CopyLZFile, LZCopy, LZDone**

MapWindowPoints 3.1

Syntax void MapWindowPoints(hwndFrom, hwndTo, lppt, cPoints)

procedure MapWindowPoints(FromWnd, ToWnd: HWnd; var Point; Count: Word);

The **MapWindowPoints** function converts (maps) a set of points from a coordinate space relative to one window to a coordinate space relative to another window.

Parameters

hwndFrom Identifies the window from which points are converted. If this parameter is NULL or HWND_DESKTOP, the points are assumed to be in screen coordinates.

hwndTo Identifies the window to which points are converted. If this parameter is NULL or HWND_DESKTOP, the points are converted to screen coordinates.

lppt Points to an array of **POINT** structures that contain the set of points to be converted. This parameter can also point to a **RECT** structure, in which case the *cPoints* parameter should be set to 2. The **POINT** structure has the following form:

```
typedef struct tagPOINT {   /* pt */
   int x;
   int y;
} POINT;
```

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

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

cPoints	Specifies the number of **POINT** structures in the array pointed to by the *lppt* parameter.

Return Value This function does not return a value.

See Also **ClientToScreen, ScreenToClient**

MemManInfo 3.1

Syntax #include <toolhelp.h>
BOOL MemManInfo(lpmmi)

function MemManInfo(lpEnhMode: PMemManInfo): Bool;

The **MemManInfo** function fills the specified structure with status and performance information about the memory manager. This function is most useful in 386 enhanced mode but can also be used in standard mode.

Parameters

lpmmi Points to a **MEMMANINFO** structure that will receive information about the memory manager. The **MEMMANINFO** structure has the following form:

```
#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;
```

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

Comments This function is included for advisory purposes.

Before calling **MemManInfo**, an application must initialize the **MEMMANINFO** structure and specify its size, in bytes, in the **dwSize** member.

MemoryRead 3.1

Syntax

#include <toolhelp.h>
DWORD MemoryRead(wSel, dwOffset, lpvBuf, dwcb)

function MemoryRead(wSel: Word; dwOffset: Longint; lpBuffer: PChar; dwcb: Longint): Longint;

The **MemoryRead** function copies memory from the specified global heap object to the specified buffer.

Parameters

wSel	Specifies the global heap object from which to read. This value must be a selector on the global heap; if the value is an alias selector or a selector in a tiled selector array, **MemoryRead** will fail.
dwOffset	Specifies the offset in the object specified in the *wSel* parameter at which to begin reading. The *dwOffset* value may point anywhere within the object; it may be greater than 64K if the object is larger than 64K.
lpvBuf	Points to the buffer to which **MemoryRead** will copy the memory from the object. This buffer must be large enough to contain the entire amount of memory copied to it. If the application is running under low memory conditions, *lpvBuf* should be in a fixed object while **MemoryRead** copies data to it.
dwcb	Specifies the number of bytes to copy from the object to the buffer pointed to by *lpvBuf.*

Return Value

The return value is the number of bytes copied from *wSel* to *lpvBuf.* If *wSel* is invalid or if *dwOffset* is out of the selector's range, the return value is zero.

Comments

The **MemoryRead** function enables developers to examine memory without consideration for selector tiling and aliasing. **MemoryRead** reads memory in read-write or read-only objects. This function can be used in any size object owned by any task. It is not necessary to compute selector array offsets.

The **MemoryRead** and **MemoryWrite** functions are designed to read and write objects loaded by the **LoadModule** function or allocated by the **GlobalAlloc** function. Developers should *not* split off the selector portion of a far pointer and use this as the value for *wSel,* unless the selector is known to be on the global heap.

See Also

MemoryWrite

MemoryWrite 3.1

Syntax #include <toolhelp.h>
DWORD MemoryWrite(wSel, dwOffset, lpvBuf, dwcb)

function MemoryWrite(wSel: Word; dwOffset: Longint; lpBuffer: PChar; dwcb: Longint): Longint;

The **MemoryWrite** function copies memory from the specified buffer to the specified global heap object.

Parameters

wSel	Specifies the global heap object to which **MemoryWrite** will write. This value must be a selector on the global heap; if the value is an alias selector or a selector in a tiled selector array, **MemoryWrite** will fail.
dwOffset	Specifies the offset in the object at which to begin writing. The *dwOffset* value may point anywhere within the object; it may be greater than 64K if the object is larger than 64K.
lpvBuf	Points to the buffer from which **MemoryWrite** will copy the memory to the object. If the application is running under low memory conditions, *lpvBuf* should be in a fixed object while **MemoryWrite** copies data from it.
dwcb	Specifies the number of bytes to copy to the object from the buffer pointed to by *lpvBuf*.

Return Value The return value is the number of bytes copied from *lpvBuf* to *wSel*. If the selector is invalid or if *dwOffset* is out of the selector's range, the return value is zero.

Comments The **MemoryWrite** function enables developers to modify memory without consideration for selector tiling and aliasing. **MemoryWrite** writes memory in read-write or read-only objects. This function can be used in any size object owned by any task. It is not necessary to make alias objects writable or to compute selector array offsets.

The **MemoryRead** and **MemoryWrite** functions are designed to read and write objects loaded by the **LoadModule** function or allocated by the **GlobalAlloc** function. Developers should *not* split off the selector portion of a far pointer and use this as the value for *wSel*, unless the selector is known to be on the global heap.

See Also **MemoryRead**

MessageProc 3.1

Syntax LRESULT CALLBACK MessageProc(code, wParam, lParam)

The **MessageProc** function is an application- or library-defined callback function that the system calls after a dialog box, message box, or menu has retrieved a message, but before the message is processed. The callback function can process or modify the messages.

Parameters

code Specifies the type of message being processed. This parameter can be one of the following values:

Value	Meaning
MSGF_DIALOGBOX	Messages inside a dialog box or message box procedure are being processed.
MSGF_MENU	Keyboard and mouse messages in a menu are being processed.

If the *code* parameter is less than zero, the callback function must pass the message to **CallNextHookEx** without further processing and return the value returned by **CallNextHookEx**.

wParam Specifies a NULL value.

lParam Points to an **MSG** structure. The **MSG** structure has the following form:

```
typedef struct tagMSG {     /* msg */
    HWND   hwnd;
    UINT   message;
    WPARAM wParam;
    LPARAM lParam;
    DWORD  time;
    POINT  pt;
} MSG;
```

Return Value The callback function should return a nonzero value if it processes the message; it should return zero if it does not process the message.

Comments The WH_MSGFILTER filter type is the only task-specific filter. A task may install this filter.

An application must install the callback function by specifying the WH_MSGFILTER filter type and the procedure-instance address of the callback function in a call to the **SetWindowsHookEx** function.

MessageProc is a placeholder for the library-defined function name. The actual name must be exported by including it in an **EXPORTS** statement in the library's module-definition file.

See Also **CallNextHookEx, SetWindowsHookEx**

ModuleFindHandle 3.1

Syntax #include <toolhelp.h>
HMODULE ModuleFindHandle(lpme, hmod)

function ModuleFindHandle(lpModule: PModuleEntry; hModule: THandle): THandle;

The **ModuleFindHandle** function fills the specified structure with information that describes the given module.

Parameters

lpme Points to a **MODULEENTRY** structure that will receive information about the module. The **MODULEENTRY** structure has the following form:

```
#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;
```

hmod Identifies the module to be described.

Return Value The return value is the handle of the given module if the function is successful. Otherwise, it is NULL.

Comments The **ModuleFindHandle** function returns information about a currently loaded module whose module handle is known.

This function can be used to begin a walk through the list of all currently loaded modules. An application can examine subsequent items in the module list by using the **ModuleNext** function.

Before calling **ModuleFindHandle**, an application must initialize the **MODULEENTRY** structure and specify its size, in bytes, in the **dwSize** member.

See Also **ModuleFindName, ModuleFirst, ModuleNext**

ModuleFindName 3.1

Syntax #include <toolhelp.h>
HMODULE ModuleFindName(lpme, lpszName)

function ModuleFindName(lpModule: PModuleEntry; lpstrName: PChar): THandle;

The **ModuleFindName** function fills the specified structure with information that describes the module with the specified name.

Parameters

lpme Points to a **MODULEENTRY** structure that will receive information about the module. The **MODULEENTRY** structure has the following form:

```
#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;
```

lpszName Specifies the name of the module to be described.

Return Value The return value is the handle named in the **lpszName** parameter, if the function is successful. Otherwise, it is NULL.

Comments The **ModuleFindName** function returns information about a currently loaded module by looking up the module's name in the module list.

This function can be used to begin a walk through the list of all currently loaded modules. An application can examine subsequent items in the module list by using the **ModuleNext** function.

Before calling **ModuleFindName**, an application must initialize the **MODULEENTRY** structure and specify its size, in bytes, in the **dwSize** member.

See Also **ModuleFindHandle, ModuleFirst, ModuleNext**

ModuleFirst 3.1

Syntax #include <toolhelp.h>
BOOL ModuleFirst(lpme)

function ModuleFirst(lpModule: PModuleEntry): Bool;

The **ModuleFirst** function fills the specified structure with information that describes the first module in the list of all currently loaded modules.

Parameters *lpme* Points to a **MODULEENTRY** structure that will receive information about the first module. The **MODULEENTRY** structure has the following form:

```
#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;
```

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

Comments The **ModuleFirst** function can be used to begin a walk through the list of all currently loaded modules. An application can examine subsequent items in the module list by using the **ModuleNext** function.

Before calling **ModuleFirst**, an application must initialize the **MODULEENTRY** structure and specify its size, in bytes, in the **dwSize** member.

See Also **ModuleFindHandle, ModuleFindName, ModuleNext**

ModuleNext 3.1

Syntax #include <toolhelp.h>
BOOL ModuleNext(lpme)

function ModuleNext(lpModule: PModuleEntry): Bool;

The **ModuleNext** function fills the specified structure with information that describes the next module in the list of all currently loaded modules.

Parameters *lpme* Points to a **MODULEENTRY** structure that will receive information about the next module. The **MODULEENTRY** structure has the following form:

```
#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;
```

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

Comments The **ModuleNext** function can be used to continue a walk through the list of all currently loaded modules. The walk must have been started by the **ModuleFirst**, **ModuleFindName**, or **ModuleFindHandle** function.

See Also **ModuleFindHandle, ModuleFindName, ModuleFirst**

MouseProc 3.1

Syntax LRESULT CALLBACK MouseProc(code, wParam, lParam)

The **MouseProc** function is a library-defined callback function that the system calls whenever an application calls the **GetMessage** or **PeekMessage** function and there is a mouse message to be processed.

Parameters

code Specifies whether the callback function should process the message or call the **CallNextHookEx** function. If this value is less than zero, the callback function should pass the message to **CallNextHookEx** without further processing. If this value is HC_NOREMOVE, the application is using a **PeekMessage** function with the PM_NOREMOVE option, and the message will not be removed from the system queue.

wParam Specifies the identifier of the mouse message.

lParam Points to a **MOUSEHOOKSTRUCT** structure containing information about the mouse. The **MOUSEHOOKSTRUCT** structure has the following form:

```
typedef struct tagMOUSEHOOKSTRUCT { /* ms */
    POINT   pt;
    HWND    hwnd;
    UINT    wHitTestCode;
    DWORD   dwExtraInfo;
} MOUSEHOOKSTRUCT;
```

The callback function should return 0 to allow the system to process the message; it should return 1 to discard the message.

Comments This callback function should not install a **JournalPlaybackProc** callback function.

An application must install the callback function by specifying the WH_MOUSE filter type and the procedure-instance address of the callback function in a call to the **SetWindowsHookEx** function.

MouseProc is a placeholder for the library-defined function name. The actual name must be exported by including it in an **EXPORTS** statement in the library's module-definition file.

See Also **CallNextHookEx, GetMessage, PeekMessage, SetWindowsHookEx**

MoveToEx 3.1

Syntax BOOL MoveToEx(hdc, nX, nY, lpPoint)

function MoveToEx(DC: HDC; nX, nY: Integer; Point: PPoint): Bool;

The **MoveToEx** function moves the current position to the point specified by the *nX* and *nY* parameters, optionally returning the previous position.

Parameters

hdc	Identifies the device context.
nX	Specifies the logical x-coordinate of the new position.
nY	Specifies the logical y-coordinate of the new position.
lpPoint	Points to a **POINT** structure in which the previous current position will be stored. If this parameter is NULL, no previous position is returned. The **POINT** structure has the following form:

```
typedef struct tagPOINT {   /* pt */
   int x;
   int y;
} POINT;
```

Return Value The return value is nonzero if the call is successful. Otherwise, it is zero.

See Also **MoveTo**

NotifyProc 2.x

Syntax BOOL CALLBACK NotifyProc(hglbl)

The **NotifyProc** function is a library-defined callback function that the system calls whenever it is about to discard a global memory object allocated with the GMEM_NOTIFY flag.

Parameters

hglbl	Identifies the global memory object being discarded.

Return Value The callback function should return nonzero if the system is to discard the memory object, or zero if it should not.

Comments The callback function is not necessarily called in the context of the application that owns the routine. For this reason, the callback function should not assume it is using the stack segment of the application. The callback function should not call any routine that might move memory.

The callback function must be in a fixed code segment of a dynamic-link library.

NotifyProc is a placeholder for the application-defined function name. The actual name must be exported by including it in an **EXPORTS** statement in the library's module-definition statement.

See Also **GlobalNotify**

NotifyRegister 3.1

Syntax

#include <toolhelp.h>
BOOL NotifyRegister(htask, lpfnCallback, wFlags)

function NotifyRegister(hTask: THandle; lpfn: TNotifyCallBack; wFlags: Word): Bool;

The **NotifyRegister** function installs a notification callback function for the given task.

Parameters

htask Identifies the task associated with the callback function. If this parameter is NULL, it identifies the current task.

lpfnCallback Points to the notification callback function that is installed for the task. The kernel calls this function whenever it sends a notification to the task.

The callback-function address is normally the return value of a call to **MakeProcInstance**. This causes the callback function to be entered with the AX register set to the selector of the application's data segment. Usually, an exported function prolog contains the following code:

```
mov  ds,ax
```

wFlags Specifies the optional notifications that the application will receive, in addition to the default notifications. This parameter can be NF_NORMAL or any combination of the following values:

Value	Meaning
NF_NORMAL	The application will receive the default notifications but none of the notifications of task switching, system debugging errors, or debug strings.
NF_TASKSWITCH	The application will receive task-switching notifications. To avoid poor performance, an application should not receive these notifications unless absolutely necessary.
NF_RIP	The application will receive notifications of system debugging errors.

Return Value The return value is nonzero if the function was successful. Otherwise, it is zero.

Callback Function The syntax of the function pointed to by *lpfnCallback* is as follows:

BOOL NotifyRegisterCallback(*wID*, *dwData*)
WORD *wID*;
DWORD *dwData*;

TNotifyCallBack = function(wID: Word; dwData: Longint): Bool;

Parameters *wID* Indicates the type of notification and the value of the *dwData* parameter. The *wID* parameter may be one of the following values in Windows versions 3.0 and later:

Value	Meaning
NFY_DELMODULE	The low-order word of *dwData* is the handle of the module to be freed.
NFY_EXITTASK	The low-order byte of *dwData* contains the program exit code.
NFY_FREESEG	The low-order word of *dwData* is the selector of the segment to be freed.
NFY_INCHAR	The *dwData* parameter is not used. The notification callback function should return either the ASCII value for the keystroke or NULL.
NFY_LOADSEG	The *dwData* parameter points to an **NFYLOADSEG** structure.
NFY_OUTSTR	The *dwData* parameter points to the string to be displayed.
NFY_RIP	The *dwData* parameter points to an **NFYRIP** structure.

Value	Meaning
NFY_STARTDLL	The *dwData* parameter points to an **NFYSTARTDLL** structure.
NFY_STARTTASK	The *dwData* parameter is the CS:IP of the starting address of the task.
NFY_UNKNOWN	The kernel returned an unknown notification.

In Windows version 3.1, *wID* may be one of the following values:

Value	Meaning
NFY_LOGERROR	The *dwData* parameter points to an **NFYLOGERROR** structure.
NFY_LOGPARAMERROR	The *dwData* parameter points to an **NFYLOGPARAMERROR** structure.
NFY_TASKIN	The *dwData* parameter is undefined. The callback function should call the **GetCurrentTask** function.
NFY_TASKOUT	The *dwData* parameter is undefined. The callback function should call **GetCurrentTask**.

dwData Specifies data, or specifies a pointer to data, or is undefined, depending on the value of *wID*.

Return Value The return value of the callback function is nonzero if the callback function handled the notification. Otherwise, it is zero and the notification is passed to other callback functions.

Comments A notification callback function must be able to ignore any unknown notification value. Typically, the notification callback function cannot use any Windows function, with the exception of the Tool Helper functions and **PostMessage**.

NotifyRegisterCallback is a placeholder for the application-defined function name. The actual name must be exported by including it in an **EXPORTS** statement in the application's module-definition file.

See Also **InterruptRegister, InterruptUnRegister, MakeProcInstance, NotifyUnRegister, TerminateApp**

NotifyUnRegister 3.1

Syntax #include <toolhelp.h>
BOOL NotifyUnRegister(htask)

function NotifyUnRegister(hTask: THandle): Bool;

The **NotifyUnRegister** function restores the default notification handler.

Parameters

htask	Identifies the task. If *htask* is NULL, it identifies the current task.

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

Comments After this function is executed, the given task no longer receives notifications from the kernel.

See Also **InterruptRegister, InterruptUnRegister, NotifyRegister, TerminateApp**

OffsetViewportOrgEx 3.1

Syntax BOOL OffsetViewportOrgEx(hdc, nX, nY, lpPoint)

function OffsetViewportOrgEx(DC: HDC; nX, nY: Integer; Point: PPoint): Bool;

The **OffsetViewportOrgEx** function modifies the viewport origin relative to the current values. The formulas are written as follows:

```
xNewVO = xOldVO + X
yNewVO = yOldVO + Y
```

The new origin is the sum of the current origin and the *nX* and *nY* values.

Parameters

hdc	Identifies the device context.
nX	Specifies the number of device units to add to the current origin's x-coordinate.
nY	Specifies the number of device units to add to the current origin's y-coordinate.
lpPoint	Points to a **POINT** structure. The previous viewport origin (in device coordinates) is placed in this structure. If *lpPoint* is NULL, the previous viewport origin in not returned.

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

OffsetWindowOrgEx 3.1

Syntax BOOL OffsetWindowOrgEx(hdc, nX, nY, lpPoint)

function OffsetWindowOrgEx(DC: HDC; nX, nY: Integer; Point: PPoint): Bool;

The **OffsetWindowOrgEx** function modifies the viewport origin relative to the current values. The formulas are written as follows:

```
xNewWO = xOldWO + X
yNewWO = yOldWO + Y
```

The new origin is the sum of the current origin and the *nX* and *nY* values.

Parameters

hdc	Identifies the device context.
nX	Specifies the number of logical units to add to the current origin's x-coordinate.
nY	Specifies the number of logical units to add to the current origin's y-coordinate.
lpPoint	Points to a **POINT** structure. The previous window origin (in logical coordinates) is placed in this structure. If *lpPoint* is NULL, the previous origin is not returned.

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

OleActivate 3.1

Syntax #include <ole.h>
OLESTATUS OleActivate(lpObject, verb, fShow, fTakeFocus, hwnd, lprcBound)

function OleActivate(Self: POleObject; Verb: Word; Show, TakeFocus: Bool; hWnd: HWnd; Bounds: PRect): TOleStatus;

The **OleActivate** function opens an object for an operation. Typically, the object is edited or played.

Parameters

lpObject	Points to the object to activate.
verb	Specifies which operation to perform (0 = the primary verb, 1 = the secondary verb, and so on).
fShow	Specifies whether the window is to be shown. If the window is to be shown, this value is TRUE; otherwise, it is FALSE.
fTakeFocus	Specifies whether the server should get the focus. If the server should get the focus, this value is TRUE; otherwise, it is FALSE. This parameter is relevant only if the *fShow* parameter is TRUE.
hwnd	Identifies the window of the document containing the object. This parameter can be NULL.
lprcBound	Points to a **RECT** structure containing the coordinates of the bounding rectangle in which the destination document displays the object. This parameter can be NULL. The mapping mode of the device context determines the units for these coordinates.

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_BUSY
OLE_ERROR_OBJECT
OLE_WAIT_FOR_RELEASE

Comments Typically, a server is launched in a separate window; editing then occurs asynchronously. The client is notified of changes to the object through the callback function.

A client application might set the *fShow* parameter to FALSE if a server needed to remain active without being visible on the display. (In this case, the application would also use the **OleSetData** function.)

Client applications typically specify the primary verb when the user double-clicks an object. The server can take any action in response to the specified verb. If the server supports only one action, it takes that action no matter which value is passed in the *verb* parameter.

In future releases of the object linking and embedding (OLE) protocol, the *hwnd* and *lprcBound* parameters will be used to help determine the placement of the server's editing window.

See Also **OleQueryOpen, OleSetData**

OleBlockServer 3.1

Syntax

#include <ole.h>
OLESTATUS OleBlockServer(lhSrvr)

function OleBlockServer(Server: LHServer): TOleStatus;

The **OleBlockServer** function causes requests to the server to be queued until the server calls the **OleUnblockServer** function.

Parameters

lhSrvr Identifies the server for which requests are to be queued.

Return Value

The return value is OLE_OK if the function is successful. Otherwise, it is an error value, which may be OLE_ERROR_HANDLE.

Comments

The server must call the **OleUnblockServer** function after calling the **OleBlockServer** function.

A server application can use the **OleBlockServer** and **OleUnblockServer** functions to control when the server library processes requests from client applications. Because only messages from the client to the server are blocked, a blocked server can continue to send messages to client applications.

A server application receives a handle when it calls the **OleRegisterServer** function.

See Also **OleRegisterServer, OleUnblockServer**

OleClone 3.1

Syntax #include <ole.h>
OLESTATUS OleClone(lpObject, lpClient, lhClientDoc, lpszObjname, lplpObject)

function OleClone(OleObject: POleObject; Client: POleClient; ClientDoc: LHClientDoc; ObjName: PChar; var OleObject: POleObject): TOleStatus;

The **OleClone** function makes a copy of an object. The copy is identical to the source object, but it is not connected to the server.

Parameters

lpObject	Points to the object to copy.
lpClient	Points to an **OLECLIENT** structure for the new object.
lhClientDoc	Identifies the client document in which the object is to be created.
lpszObjname	Points to a null-terminated string specifying the client's name for the object. This name must be unique with respect to the names of any other objects in the document and cannot contain a slash mark (/).
lplpObject	Points to a variable where the library will store the long pointer to the new object.

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_BUSY
OLE_ERROR_HANDLE
OLE_ERROR_OBJECT
OLE_WAIT_FOR_RELEASE

Comments Client applications often use the **OleClone** function to support the Undo command.

A client application can supply a new **OLECLIENT** structure for the cloned object, if required.

See Also **OleEqual**

OleClose 3.1

Syntax #include <ole.h>
OLESTATUS OleClose(lpObject)

function OleClose(Self: POleObject): TOleStatus;

The **OleClose** function closes the specified open object. Closing an object terminates the connection with the server application.

Parameters

lpObject	Points to the object to close.

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_BUSY
OLE_ERROR_OBJECT
OLE_WAIT_FOR_RELEASE

See Also **OleActivate, OleDelete, OleReconnect**

OleCopyFromLink 3.1

Syntax #include <ole.h>
OLESTATUS OleCopyFromLink(lpObject, lpszProtocol, lpClient, lhClientDoc, lpszObjname, lplpObject)

function OleCopyFromLink(OleObject: POleObject; Protocol: PChar; Client: POleClient; ClientDoc: LHClientDoc; ObjName: PChar; var OleObject: POleObject): TOleStatus;

The **OleCopyFromLink** function makes an embedded copy of a linked object.

Parameters

lpObject	Points to the linked object that is to be embedded.
lpszProtocol	Points to a null-terminated string specifying the name of the protocol required for the new embedded object. Currently, this value can be StdFileEditing (the name of the object linking and embedding protocol).
lpClient	Points to an **OLECLIENT** structure for the new object.
lhClientDoc	Identifies the client document in which the object is to be created.

lpszObjname	Points to a null-terminated string specifying the client's name for the object.
lplpObject	Points to a variable where the long pointer to the new object will be stored.

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_BUSY
OLE_ERROR_HANDLE
OLE_ERROR_NAME
OLE_ERROR_OBJECT
OLE_ERROR_PROTOCOL
OLE_WAIT_FOR_RELEASE

Comments Making an embedded copy of a linked object may involve starting the server application.

See Also **OleObjectConvert**

OleCopyToClipboard 3.1

Syntax

```
#include <ole.h>
OLESTATUS OleCopyToClipboard(lpObject)

function OleCopyToClipboard(Self: POleObject): TOleStatus;
```

The **OleCopyToClipboard** function puts the specified object on the clipboard.

Parameters

lpObject	Points to the object to copy to the clipboard.

Return Value The return value is OLE_OK if the function is successful. Otherwise, it is an error value, which may be OLE_ERROR_OBJECT.

Comments A client application typically calls the **OleCopyToClipboard** function when a user chooses the Copy or Cut command from the Edit menu.

The client application should open and empty the clipboard, call the **OleCopyToClipboard** function, and close the clipboard.

OleCreate 3.1

Syntax #include <ole.h>
OLESTATUS OleCreate(lpszProtocol, lpClient, lpszClass, lhClientDoc, lpszObjname, lplpObject, renderopt, cfFormat)

function OleCreate(Protocol: PChar; Client: POleClient; Class: PChar; ClientDoc: LHClientDoc; ObjectName: PChar; var OleObject: POleObject; RenderOpt: TOleOPT_Render; Format: TOleClipFormat): TOleStatus;

The **OleCreate** function creates an embedded object of a specified class. The server is opened to perform the initial editing.

Parameters

lpszProtocol Points to a null-terminated string specifying the name of the protocol required for the new embedded object. Currently, this value can be StdFileEditing (the name of the object linking and embedding protocol).

lpClient Points to an **OLECLIENT** structure for the new object.

lpszClass Points to a null-terminated string specifying the registered name of the class of the object to be created.

lhClientDoc Identifies the client document in which the object is to be created.

lpszObjname Points to a null-terminated string specifying the client's name for the object. This name must be unique with respect to the names of any other objects in the document and cannot contain a slash mark (/).

lplpObject Points to a variable where the library will store the long pointer to the new object.

renderopt Specifies the client's preference for presentation data for the object. This parameter can be one of the following values:

Value	Meaning
olerender_draw	The client calls the **OleDraw** function, and the library obtains and manages presentation data.
olerender_format	The client calls the **OleGetData** function to retrieve data in a specific format. The library obtains and manages the data in the requested format, as specified by the *cfFormat* parameter.

Value	Meaning
olerender_none	The client library does not obtain any presentation data and does not draw the object.

Return Value *cfFormat* Specifies the clipboard format when the *renderopt* parameter is **olerender_format**. This clipboard format is used in a subsequent call to **OleGetData**. If this clipboard format is CF_METAFILEPICT, CF_DIB, or CF_BITMAP, the library manages the data and draws the object. The library does not support drawing for any other formats. 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_HANDLE
OLE_ERROR_NAME
OLE_ERROR_PROTOCOL
OLE_WAIT_FOR_RELEASE

Comments The **olerender_none** rendering option is typically used to support hyperlinks. With this option, the client does not call **OleDraw** and calls **OleGetData** only for ObjectLink, OwnerLink, and Native formats.

The **olerender_format** rendering option allows a client to compute data (instead of painting it), use an unusual data format, or modify a standard data format. With this option, the client does not call **OleDraw**. The client calls **OleGetData** to retrieve data in the specified format.

The **olerender_draw** rendering option is the most typical option. It is the easiest rendering option for the client to implement (the client simply calls **OleDraw**), and it allows the most flexibility. An object handler can exploit this flexibility to store no presentation data, a private presentation data format, or several different formats that it can choose among dynamically. Future implementations of object linking and embedding (OLE) may also exploit the flexibility that is inherent in this option.

See Also **OleCreateFromClip**, **OleCreateFromTemplate**, **OleDraw**, **OleGetData**

OleCreateFromClip 3.1

Syntax #include <ole.h>
OLESTATUS OleCreateFromClip(lpszProtocol, lpClient, lhClientDoc, lpszObjname, lplpObject, renderopt, cfFormat)

function OleCreateFromClip(Protocol: PChar; Client: POleClient; ClientDoc: LHClientDoc; ObjName: PChar; var OleObject: POleObject; RenderOpt: TOleOPT_Render; Format: TOleClipformat): TOleStatus;

The **OleCreateFromClip** function creates an object from the clipboard.

Parameters

lpszProtocol Points to a null-terminated string specifying the name of the protocol required for the new embedded object. Currently, this value can be StdFileEditing (the name of the object linking and embedding protocol) or Static (for uneditable pictures only).

lpClient Points to an **OLECLIENT** structure allocated and initialized by the client application. This pointer is used to locate the callback function and is passed in callback notifications.

lhClientDoc Identifies the client document in which the object is being created.

lpszObjname Points to a null-terminated string specifying the client's name for the object. This name must be unique with respect to the names of any other objects in the document and cannot contain a slash mark (/).

lplpObject Points to a variable where the library will store the long pointer to the new object.

renderopt Specifies the client's preference for presentation data for the object. This parameter can be one of the following values:

Value	Meaning
olerender_draw	The client calls the **OleDraw** function, and the library obtains and manages presentation data.
olerender_format	The client calls the **OleGetData** function to retrieve data in a specific format. The library obtains and manages the data in the requested format, as specified by the *cfFormat* parameter.
olerender_none	The client library does not obtain any presentation data and does not draw the object.

cfFormat Specifies the clipboard format when the *renderopt* parameter is **olerender_format**. This clipboard format is used in a subsequent call to **OleGetData**. If this clipboard format is CF_METAFILEPICT, CF_DIB, or CF_BITMAP, the library manages the data and draws the object. The library does not support drawing for any other formats.

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_CLIP
OLE_ERROR_FORMAT
OLE_ERROR_HANDLE
OLE_ERROR_NAME
OLE_ERROR_OPTION
OLE_ERROR_PROTOCOL
OLE_WAIT_FOR_RELEASE

Comments The client application should open and empty the clipboard, call the **OleCreateFromClip** function, and close the clipboard.

The **olerender_none** rendering option is typically used to support hyperlinks. With this option, the client does not call **OleDraw** and calls **OleGetData** only for ObjectLink, OwnerLink, and Native formats.

The **olerender_format** rendering option allows a client to compute data (instead of painting it), use an unusual data format, or modify a standard data format. With this option, the client does not call **OleDraw**. The client calls **OleGetData** to retrieve data in the specified format.

The **olerender_draw** rendering option is the most typical option. It is the easiest rendering option for the client to implement (the client simply calls **OleDraw**), and it allows the most flexibility. An object handler can exploit this flexibility to store no presentation data, a private presentation data format, or several different formats that it can choose among dynamically. Future implementations of object linking and embedding (OLE) may also exploit the flexibility that is inherent in this option.

See Also **OleCreate, OleCreateFromTemplate, OleDraw, OleGetData, OleQueryCreateFromClip**

OleCreateFromFile 3.1

Syntax #include <ole.h>
OLESTATUS OleCreateFromFile(lpszProtocol, lpClient, lpszClass, lpszFile, lhClientDoc, lpszObjname, lplpObject, renderopt, cfFormat)

function OleCreateFromFile(Protocol: PChar; Client: POleClient; Class, OleFile: PChar; ClientDoc: LHClientDoc; ObjName: PChar; var OleObject: PoleObject; RenderOpt: TOleOPT_Render; Format: TOleClipFormat): TOleStatus;

The **OleCreateFromFile** function creates an embedded object from the contents of a named file.

Parameters

lpszProtocol — Points to a null-terminated string specifying the name of the protocol required for the new embedded object. Currently, this value can be StdFileEditing (the name of the object linking and embedding protocol).

lpClient — Points to an **OLECLIENT** structure allocated and initialized by the client application. This pointer is used to locate the callback function and is passed in callback notifications.

lpszClass — Points to a null-terminated string specifying the name of the class for the new object. If this value is NULL, the library uses the extension of the filename pointed to by the *lpszFile* parameter to find the class name for the object.

lpszFile — Points to a null-terminated string specifying the name of the file containing the object.

lhClientDoc — Identifies the client document in which the object is being created.

lpszObjname — Points to a null-terminated string specifying the client's name for the object. This name must be unique with respect to the names of any other objects in the document and cannot contain a slash mark (/).

lplpObject — Points to a variable where the library will store the long pointer to the new object.

renderopt — Specifies the client's preference for presentation data for the object. This parameter can be one of the following values:

Value	Meaning
olerender_draw	The client calls the **OleDraw** function, and the library obtains and manages presentation data.
olerender_format	The client calls the **OleGetData** function to retrieve data in a specific format. The library obtains and manages the data in the requested format, as specified by the *cfFormat* parameter.

Value	Meaning
olerender_none	The client library does not obtain any presentation data and does not draw the object.

cfFormat Specifies the clipboard format when the *renderopt* parameter is **olerender_format**. This clipboard format is used in a subsequent call to **OleGetData**. If this clipboard format is CF_METAFILEPICT, CF_DIB, or CF_BITMAP, the library manages the data and draws the object. The library does not support drawing for any other formats.

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_CLASS
OLE_ERROR_HANDLE
OLE_ERROR_MEMORY
OLE_ERROR_NAME
OLE_ERROR_PROTOCOL
OLE_WAIT_FOR_RELEASE

Comments When a client application calls the **OleCreateFromFile** function, the server is started to render the Native and presentation data and then is closed. (If the server and document are already open, this function simply retrieves the information, without closing the server.) The server does not show the object to the user for editing.

The **olerender_none** rendering option is typically used to support hyperlinks. With this option, the client does not call **OleDraw** and calls **OleGetData** only for ObjectLink, OwnerLink, and Native formats.

The **olerender_format** rendering option allows a client to compute data (instead of painting it), use an unusual data format, or modify a standard data format. With this option, the client does not call **OleDraw**. The client calls **OleGetData** to retrieve data in the specified format.

The **olerender_draw** rendering option is the most typical option. It is the easiest rendering option for the client to implement (the client simply calls **OleDraw**), and it allows the most flexibility. An object handler can exploit this flexibility to store no presentation data, a private presentation data format, or several different formats that it can choose among dynamically. Future implementations of object linking and embedding (OLE) may also exploit the flexibility that is inherent in this option.

If a client application accepts files dropped from File Manager, it should respond to the **WM_DROPFILES** message by calling **OleCreateFromFile** and specifying Packager for the *lpszClass* parameter to indicate Microsoft Windows Object Packager.

See Also **OleCreate, OleCreateFromTemplate, OleDraw, OleGetData**

OleCreateFromTemplate 3.1

Syntax #include <ole.h>
OLESTATUS OleCreateFromTemplate(lpszProtocol, lpClient, lpszTemplate, lhClientDoc, lpszObjname, lplpObject, renderopt, cfFormat)

function OleCreateFromTemplate(Protocol: PChar; Client: POleClient; Template: PChar; ClientDoc: LHClientDoc; ObjName: PChar; var OleObject: POleObject; RenderOpt: TOleOPT_Render; Format: TOleClipFormat): TOleStatus;

The **OleCreateFromTemplate** function creates an object by using another object as a template. The server is opened to perform the initial editing.

Parameters

lpszProtocol	Points to a null-terminated string specifying the name of the protocol required for the new embedded object. Currently, this value can be StdFileEditing (the name of the object linking and embedding protocol).
lpClient	Points to an **OLECLIENT** structure for the new object.
lpszTemplate	Points to a null-terminated string specifying the path of the file to be used as a template for the new object. The server is opened for editing and loads the initial state of the new object from the named template file.
lhClientDoc	Identifies the client document in which the object is being created.
lpszObjname	Points to a null-terminated string specifying the client's name for the object. This name must be unique with respect to the names of any other objects in the document and cannot contain a slash mark (/).
lplpObject	Points to a variable where the library will store the long pointer to the new object.
renderopt	Specifies the client's preference for presentation data for the object. This parameter can be one of the following values:

Value	Meaning
olerender_draw	The client calls the **OleDraw** function, and the library obtains and manages presentation data.
olerender_format	The client calls the **OleGetData** function to retrieve data in a specific format. The library obtains and manages the data in the requested format, as specified by the *cfFormat* parameter.
olerender_none	The client library does not obtain any presentation data and does not draw the object.

cfFormat Specifies the clipboard format when the *renderopt* parameter is **olerender_format**. This clipboard format is used in a subsequent call to the **OleGetData** function. If this clipboard format is CF_METAFILEPICT, CF_DIB, or CF_BITMAP, the library manages the data and draws the object. The library does not support drawing for any other formats.

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_CLASS
OLE_ERROR_HANDLE
OLE_ERROR_MEMORY
OLE_ERROR_NAME
OLE_ERROR_PROTOCOL
OLE_WAIT_FOR_RELEASE

Comments The client library uses the filename extension of the file specified in the *lpszTemplate* parameter to identify the server for the object. The association between the extension and the server is stored in the registration database.

The **olerender_none** rendering option is typically used to support hyperlinks. With this option, the client does not call **OleDraw** and calls **OleGetData** only for ObjectLink, OwnerLink, and Native formats.

The **olerender_format** rendering option allows a client to compute data (instead of painting it), use an unusual data format, or modify a standard data format. With this option, the client does not call **OleDraw**. The client calls **OleGetData** to retrieve data in the specified format.

The **olerender_draw** rendering option is the most typical option. It is the easiest rendering option for the client to implement (the client simply calls **OleDraw**), and it allows the most flexibility. An object handler can exploit this flexibility to store no presentation data, a private presentation data format, or several different formats that it can choose among dynamically. Future implementations of object linking and embedding (OLE) may also exploit the flexibility that is inherent in this option.

See Also **OleCreate, OleCreateFromClip, OleDraw, OleGetData, OleObjectConvert**

OleCreateInvisible 3.1

Syntax

```
#include <ole.h>
OLESTATUS OleCreateInvisible(lpszProtocol, lpClient, lpszClass,
lhClientDoc, lpszObjname, lplpObject, renderopt, cfFormat, fActivate)
```

```
function OleCreateInvisible(Protocol: PChar; Client: POleClient; Class:
PChar; ClientDoc: LHClientDoc; ObjName: PChar; var OleObject:
POleObject; RenderOpt: TOleOPT_Render; Format: TOleClipFormat;
Activate: Bool): TOleStatus;
```

The **OleCreateInvisible** function creates an object without displaying the server application to the user. The function either starts the server to create the object or creates a blank object of the specified class and format without starting the server.

Parameters

lpszProtocol	Points to a null-terminated string specifying the name of the protocol required for the new embedded object. Currently, this value can be StdFileEditing (the name of the object linking and embedding protocol) or Static (for uneditable pictures only).
lpClient	Points to an **OLECLIENT** structure allocated and initialized by the client application. This pointer is used to locate the callback function and is passed in callback notifications.
lpszClass	Points to a null-terminated string specifying the registered name of the class of the object to be created.
lhClientDoc	Identifies the client document in which the object is being created.
lpszObjname	Points to a null-terminated string specifying the client's name for the object. This name must be unique with respect to the names of any other objects in the document and cannot contain a slash mark (/).

lplpObject Points to a variable where the library will store the long pointer to the new object.

renderopt Specifies the client's preference for presentation data for the object. This parameter can be one of the following values:

Value	Meaning
olerender_draw	The client calls the **OleDraw** function, and the library obtains and manages presentation data.
olerender_format	The client calls the **OleGetData** function to retrieve data in a specific format. The library obtains and manages the data in the requested format, as specified by the *cfFormat* parameter.
olerender_none	The client library does not obtain any presentation data and does not draw the object.

cfFormat Specifies the clipboard format when the *renderopt* parameter is **olerender_format**. This clipboard format is used in a subsequent call to **OleGetData**. If this clipboard format is CF_METAFILEPICT, CF_DIB, or CF_BITMAP, the library manages the data and draws the object. The library does not support drawing for any other formats.

fActivate Specifies whether to start the server for the object. If this parameter is TRUE the server is started (but not shown). If this parameter is FALSE, the server is not started and the function creates a blank object of the specified class and format.

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_HANDLE
OLE_ERROR_NAME
OLE_ERROR_PROTOCOL

Comments An application can avoid redrawing an object repeatedly by calling the **OleCreateInvisible** function before using such functions as **OleSetBounds**, **OleSetColorScheme**, and **OleSetTargetDevice** to set up the object. After setting up the object, the application can either call the **OleActivate** function to display the object or call the **OleUpdate** and **OleClose** functions to update the object without displaying it.

See Also **OleActivate, OleClose, OleSetBounds, OleSetColorScheme, OleSetTargetDevice, OleUpdate**

OleCreateLinkFromClip 3.1

Syntax #include <ole.h>
OLESTATUS OleCreateLinkFromClip(lpszProtocol, lpClient, lhClientDoc, lpszObjname, lplpObject, renderopt, cfFormat)

function OleCreateLinkFromClip(Protocol: PChar; Client: POleClient; ClientDoc: LHClientDoc; ObjectName: PChar; var OleObject: POleObject; RenderOpt: TOleOPT_Render; Format: TOleClipFormat): TOleStatus;

The **OleCreateLinkFromClip** function typically creates a link to an object from the clipboard.

Parameters

lpszProtocol	Points to a null-terminated string specifying the name of the required protocol. Currently, this value can be StdFileEditing (the name of the object linking and embedding protocol).
lpClient	Points to an **OLECLIENT** structure allocated and initialized by the client application. This pointer is used to locate the callback function and is passed in callback notifications.
lhClientDoc	Identifies the client document in which the object is being created.
lpszObjname	Points to a null-terminated string specifying the client's name for the object. This name must be unique with respect to the names of any other objects in the document and cannot contain a slash mark (/).

lplpObject Points to a variable where the library will store the long pointer to the new object.

renderopt Specifies the client's preference for presentation data for the object. This parameter can be one of the following values:

Value	Meaning
olerender_draw	The client calls the **OleDraw** function, and the library obtains and manages presentation data.
olerender_format	The client calls the **OleGetData** function to retrieve data in a specific format. The library obtains and manages the data in the requested format, as specified by the *cfFormat* parameter.
olerender_none	The client library does not obtain any presentation data and does not draw the object.

cfFormat Specifies the clipboard format when the *renderopt* parameter is **olerender_format**. This clipboard format is used in a subsequent call to **OleGetData**. If this clipboard format is CF_METAFILEPICT, CF_DIB, or CF_BITMAP, the library manages the data and draws the object. The library does not support drawing for any other formats.

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_CLIP
OLE_ERROR_FORMAT
OLE_ERROR_HANDLE
OLE_ERROR_NAME
OLE_ERROR_PROTOCOL
OLE_WAIT_FOR_RELEASE

Comments The **olerender_none** rendering option is typically used to support hyperlinks. With this option, the client does not call the **OleDraw** function and calls **OleGetData** only for ObjectLink, OwnerLink, and Native formats.

The **olerender_format** rendering option allows a client to compute data (instead of painting it), use an unusual data format, or modify a standard data format. With this option, the client does not call **OleDraw**. The client calls **OleGetData** to retrieve data in the specified format.

The **olerender_draw** rendering option is the most typical option. It is the easiest rendering option for the client to implement (the client simply calls **OleDraw**), and it allows the most flexibility. An object handler can exploit this flexibility to store no presentation data, a private presentation data format, or several different formats that it can choose among dynamically. Future implementations of object linking and embedding (OLE) may also exploit the flexibility that is inherent in this option.

See Also **OleCreate, OleCreateFromTemplate, OleDraw, OleGetData, OleQueryLinkFromClip**

OleCreateLinkFromFile 3.1

Syntax

```
#include <ole.h>
OLESTATUS OleCreateLinkFromFile(lpszProtocol, lpClient, lpszClass,
lpszFile, lpszItem, lhClientDoc, lpszObjname, lplpObject, renderopt,
cfFormat)
```

```
function OleCreateLinkFromFile(Protocol: PChar; Client: POleClient;
Class, OleFile, Item: PChar; ClientDoc: LHClientDoc; ObjName: PChar;
var OleObject: POleObject; RenderOpt: TOleOPT_Render; Format:
TOleClipFormat): TOleStatus;
```

The **OleCreateLinkFromFile** function creates a linked object from a file that contains an object. If necessary, the library starts the server to render the presentation data, but the object is not shown in the server for editing.

Parameters

lpszProtocol	Points to a null-terminated string specifying the name of the required protocol. Currently, this value can be StdFileEditing (the name of the object linking and embedding protocol).
lpClient	Points to an **OLECLIENT** structure allocated and initialized by the client application. This pointer is used to locate the callback function and is passed in callback notifications.
lpszClass	Points to a null-terminated string specifying the name of the class for the new object. If this value is NULL, the library uses the extension of the filename pointed to by the *lpszFile* parameter to find the class name for the object.
lpszFile	Points to a null-terminated string specifying the name of the file containing the object.
lpszItem	Points to a null-terminated string identifying the part of the document to link to. If this value is NULL, the link is to the entire document.

lhClientDoc	Identifies the client document in which the object is being created.
lpszObjname	Points to a null-terminated string specifying the client's name for the object. This name must be unique with respect to the names of any other objects in the document and cannot contain a slash mark (/).
lplpObject	Points to a variable where the library will store the long pointer to the new object.
renderopt	Specifies the client's preference for presentation data for the object. This parameter can be one of the following values:

Value	Meaning
olerender_draw	The client calls the **OleDraw** function, and the library obtains and manages presentation data.
olerender_format	The client calls the **OleGetData** function to retrieve data in a specific format. The library obtains and manages the data in the requested format, as specified by the *cfFormat* parameter.
olerender_none	The client library does not obtain any presentation data and does not draw the object.

cfFormat	Specifies the clipboard format when the *renderopt* parameter is **olerender_format**. This clipboard format is used in a subsequent call to **OleGetData**. If this clipboard format is CF_METAFILEPICT, CF_DIB, or CF_BITMAP, the library manages the data and draws the object. The library does not support drawing for any other formats.

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_CLASS
OLE_ERROR_HANDLE
OLE_ERROR_MEMORY
OLE_ERROR_NAME
OLE_ERROR_PROTOCOL
OLE_WAIT_FOR_RELEASE

Comments The **olerender_none** rendering option is typically used to support hyperlinks. With this option, the client does not call **OleDraw** and calls **OleGetData** only for ObjectLink, OwnerLink, and Native formats.

The **olerender_format** rendering option allows a client to compute data (instead of painting it), use an unusual data format, or modify a standard data format. With this option, the client does not call **OleDraw**. The client calls **OleGetData** to retrieve data in the specified format.

The **olerender_draw** rendering option is the most typical option. It is the easiest rendering option for the client to implement (the client simply calls **OleDraw**), and it allows the most flexibility. An object handler can exploit this flexibility to store no presentation data, a private presentation data format, or several different formats that it can choose among dynamically. Future implementations of object linking and embedding (OLE) may also exploit the flexibility that is inherent in this option.

See Also **OleCreate, OleCreateFromFile, OleCreateFromTemplate, OleDraw, OleGetData**

OleDelete 3.1

Syntax

```
#include <ole.h>
OLESTATUS OleDelete(lpObject)
```

```
function OleDelete(Self: POleObject): TOleStatus;
```

The **OleDelete** function deletes an object and frees memory that was associated with that object. If the object was open, it is closed.

Parameters

lpObject	Points to the object to delete.

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_BUSY
OLE_ERROR_OBJECT
OLE_WAIT_FOR_RELEASE

Comments An application uses the **OleDelete** function when the object is no longer part of the client document.

The **OleDelete** function, unlike **OleRelease**, indicates that the object has been permanently removed.

See Also **OleClose, OleRelease**

OleDraw 3.1

Syntax

#include <ole.h>
OLESTATUS OleDraw(lpObject, hdc, lprcBounds, lprcWBounds, hdcFormat)

function OleDraw(Self: POleObject; DC: HDC; var Bounds, WBounds, TRect; FormatDC: HDC): TOleStatus;

The **OleDraw** function draws a specified object into a bounding rectangle in a device context.

Parameters

lpObject	Points to the object to draw.
hdc	Identifies the device context in which to draw the object.
lprcBounds	Points to a **RECT** structure defining the bounding rectangle, in logical units for the device context specified by the *hdc* parameter, in which to draw the object.
lprcWBounds	Points to a **RECT** structure defining the bounding rectangle if the *hdc* parameter specifies a metafile. The **left** and **top** members of the **RECT** structure should specify the window origin, and the **right** and **bottom** members should specify the window extents.
hdcFormat	Identifies a device context describing the target device for which to format the object.

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_ABORT
OLE_ERROR_BLANK
OLE_ERROR_DRAW
OLE_ERROR_MEMORY
OLE_ERROR_OBJECT

Comments

This function returns OLE_ERROR_ABORT if the callback function returns FALSE during drawing.

When the *hdc* parameter specifies a metafile device context, the rectangle specified by the *lprcWBounds* parameter contains the rectangle specified by the *lprcBounds* parameter. If *hdc* does not specify a metafile device context, the *lprcWBounds* parameter is ignored.

The library may use an object handler to render the object, and this object handler may need information about the target device. Therefore, the device-context handle specified by the *hdcFormat* parameter is required. The *lprcBounds* parameter identifies the rectangle on the device context (relative to its current mapping mode) that the object should be mapped onto. This may involve scaling the picture and can be used by client applications to impose a view scaling between the displayed view and the final printed image.

An object handler should format an object as if it were to be drawn at the size specified by a call to the **OleSetBounds** function for the device context specified by the *hdcFormat* parameter. Often this formatting will already have been done by the server application; in this case, the library simply renders the presentation data with suitable scaling for the required bounding rectangle. If cropping or banding is required, the device context in which the object is drawn may include a clipping region smaller than the specified bounding rectangle.

See Also **OleSetBounds**

OleEnumFormats 3.1

Syntax

```
#include &lt;ole.h&gt;
OLECLIPFORMAT OleEnumFormats(lpObject, cfFormat)
```

function OleEnumFormats(Self: POleObject; Format: TOleClipFormat): TOleClipFormat;

The **OleEnumFormats** function enumerates the data formats that describe a specified object.

Parameters

lpObject	Points to the object to be queried.
cfFormat	Specifies the format returned by the last call to the **OleEnumFormats** function. For the first call to this function, this parameter is zero.

Return Value The return value is the next available format if any further formats are available. Otherwise, the return value is NULL.

Comments When an application specifies NULL for the *cfFormat* parameter, the **OleEnumFormats** function returns the first available format. Whenever an application specifies a format that was returned by a previous call to **OleEnumFormats**, the function returns the next available format, in sequence. When no more formats are available, the function returns NULL.

See Also **OleGetData**

OleEnumObjects 3.1

Syntax #include <ole.h>
OLESTATUS OleEnumObjects(lhDoc, lplpObject)

function OleEnumObjects(ClientDoc: LHClientDoc; var OleObject: POleObject): TOleStatus;

The **OleEnumObjects** function enumerates the objects in a specified document.

Parameters

lhDoc	Identifies the document for which the objects are enumerated.
lplpObject	Points to an object in the document when the function returns. For the first call to this function, this parameter should point to a NULL object.

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_HANDLE
OLE_ERROR_OBJECT

Comments When an application specifies a NULL object for the *lplpObject* parameter, the **OleEnumObjects** function returns the first object in the document. Whenever an application specifies an object that was returned by a previous call to **OleEnumObjects**, the function returns the next object, in sequence. When there are no more objects in the document, the *lplpObject* parameter points to a NULL object.

Only objects that have been loaded and not released are enumerated by this function.

See Also **OleDelete, OleRelease**

OleEqual 3.1

Syntax #include <ole.h>
OLESTATUS OleEqual(lpObject1, lpObject2)

function OleEqual(Self: POleObject; OleObject: POleObject): TOleStatus;

The **OleEqual** function compares two objects for equality.

Parameters

lpObject1	Points to the first object to test for equality.
lpObject2	Points to the second object to test for equality.

Return Value The return value is OLE_OK if the specified objects are equal. Otherwise, it is an error value, which may be one of the following:

OLE_ERROR_OBJECT
OLE_ERROR_NOT_EQUAL

Comments Embedded objects are equal if their class, item, and native data are identical. Linked objects are equal if their class, document, and item are identical.

See Also **OleClone, OleQueryOutOfDate**

OleExecute 3.1

Syntax #include <ole.h>
OLESTATUS OleExecute(lpObject, hglbCmds, reserved)

function OleExecute(Self: POleObject; Commands: THandle; Reserved: Word): TOleStatus;

The **OleExecute** function sends dynamic data exchange (DDE) execute commands to the server for the specified object.

Parameters

lpObject	Points to an object identifying the server to which DDE execute commands are sent.
hglbCmds	Identifies the memory containing one or more DDE execute commands.
reserved	Reserved; must be zero.

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_BUSY
OLE_ERROR_COMMAND
OLE_ERROR_MEMORY
OLE_ERROR_NOT_OPEN
OLE_ERROR_OBJECT
OLE_ERROR_PROTOCOL
OLE_ERROR_STATIC
OLE_WAIT_FOR_RELEASE

Comments The client application should call the **OleQueryProtocol** function, specifying StdExecute, before calling the **OleExecute** function. The **OleQueryProtocol** function succeeds if the server for an object supports the **OleExecute** function.

See Also **OleQueryProtocol**

OleGetData 3.1

Syntax #include <ole.h>
OLESTATUS OleGetData(lpObject, cfFormat, lphData)

function OleGetData(Self: POleObject; Format: TOleClipFormat; var Data: THandle): TOleStatus;

The **OleGetData** function retrieves data in the requested format from the specified object and supplies the handle of a memory or graphics device interface (GDI) object containing the data.

Parameters

lpObject	Points to the object from which data is retrieved.
cfFormat	Specifies the format in which data is returned. This parameter can be one of the predefined clipboard formats or the value returned by the **RegisterClipboardFormat** function.
lphData	Points to the handle of a memory object that contains the data when the function returns.

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
OLE_WARN_DELETE_DATA

Comments If the **OleGetData** function returns OLE_WARN_DELETE_DATA, the client application owns the data and should free the memory associated with the data when the client has finished using it. For other return values, the client should not free the memory or modify the data, because the data is controlled by the client library. If the application needs the data for long-term use, it should copy the data.

The **OleGetData** function typically returns OLE_WARN_DELETE_DATA if an object handler generates data for an object that the client library cannot interpret. In this case, the client application is responsible for controlling that data.

When the **OleGetData** function specifies CF_METAFILE or CF_BITMAP, the *lphData* parameter points to a GDI object, not a memory object, when the function returns. **OleGetData** supplies the handle of a memory object for all other formats.

See Also **OleEnumFormats, OleSetData, RegisterClipboardFormat**

OleGetLinkUpdateOptions 3.1

Syntax

```
#include <ole.h>
OLESTATUS OleGetLinkUpdateOptions(lpObject, lpUpdateOpt)
```

```
function OleGetLinkUpdateOptions(Self: POleObject; var UpdateOpt:
TOleOpt_Update): TOleStatus;
```

The **OleGetLinkUpdateOptions** function retrieves the link-update options for the presentation of a specified object.

Parameters

lpObject	Points to the object to query.
lpUpdateOpt	Points to a variable in which the function stores the current value of the link-update option for the specified object. The link-update option setting may be one of the following values:

Value	Meaning
oleupdate_always	Update the linked object whenever possible. This option supports the Automatic link-update radio button in the Links dialog box.
oleupdate_oncall	Update the linked object only on request from the client application. This option supports the Manual link-update radio button in the Links dialog box.
oleupdate_onsave	Update the linked object when the source document is saved by the server.

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_OBJECT
OLE_ERROR_STATIC

See Also **OleSetLinkUpdateOptions**

OleIsDcMeta 3.1

Syntax

```
#include <ole.h>
BOOL OleIsDcMeta(hdc)
```

```
function OleIsDcMeta(DC: HDC): Bool;
```

The **OleIsDcMeta** function determines whether the specified device context is a metafile device context.

Parameters

hdc	Identifies the device context to query.

Return Value The return value is a positive value if the device context is a metafile device context. Otherwise, it is NULL.

OleLoadFromStream 3.1

Syntax #include <ole.h>
OLESTATUS OleLoadFromStream(lpStream, lpszProtocol, lpClient, lhClientDoc, lpszObjname, lplpObject)

function OleLoadFromStream(Stream: POleStream; Protocol: PChar; Client: POleClient; ClientDoc: LHClientDoc; ObjectName: PChar; var OleObject: POleObject): TOleStatus;

The **OleLoadFromStream** function loads an object from the containing document.

Parameters

lpStream	Points to an **OLESTREAM** structure that was allocated and initialized by the client application. The library calls the **Get** function in the **OLESTREAMVTBL** structure to obtain the data for the object.
lpszProtocol	Points to a null-terminated string specifying the name of the required protocol. Currently, this value can be StdFileEditing (the name of the object linking and embedding protocol) or Static (for uneditable pictures only).
lpClient	Points to an **OLECLIENT** structure allocated and initialized by the client application. This pointer is used to locate the callback function and is passed in callback notifications.
lhClientDoc	Identifies the client document in which the object is being created.
lpszObjname	Points to a null-terminated string specifying the client's name for the object.
lplpObject	Points to a variable in which the library stores a pointer to the loaded object.

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_HANDLE
OLE_ERROR_NAME
OLE_ERROR_PROTOCOL
OLE_ERROR_STREAM
OLE_WAIT_FOR_RELEASE

Comments To load an object, the client application needs only the location of that object in a file. A client typically loads an object only when the object is needed (for example, when it must be displayed).

If an object cannot be loaded when the *lpszProtocol* parameter specifies StdFileEditing, the application can call the **OleLoadFromStream** function again, specifying Static.

If the object is linked and the server and document are open, the library automatically makes the link between the client and server applications when an application calls **OleLoadFromStream**.

See Also **OleQuerySize, OleSaveToStream**

OleLockServer 3.1

Syntax #include <ole.h>
OLESTATUS OleLockServer(lpObject, lphServer)

function OleLockServer(OleObject: POleObject; var Server: LHServer): TOleStatus;

The **OleLockServer** function is called by a client application to keep an open server application in memory. Keeping the server application in memory allows the client library to use the server application to open objects quickly.

Parameters

lpObject	Points to an object the client library uses to identify the open server application to keep in memory. When the server has been locked, this object can be deleted.
lphServer	Points to the handle of the server application when the function returns.

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_COMM
OLE_ERROR_LAUNCH
OLE_ERROR_OBJECT

Comments A client calls **OleLockServer** to speed the opening of objects when the same server is used for a number of different objects. Before the client terminates, it must call the **OleUnlockServer** function to release the server from memory.

When **OleLockServer** is called more than once for a given server, even by different client applications, the server's lock count is increased. Each call to **OleUnlockServer** decrements the lock count. The server remains locked until the lock count is zero. If the object identified by the *lpObject* parameter is deleted before calling the **OleUnlockServer** function, **OleUnlockServer** must still be called to decrement the lock count.

If necessary, a server can terminate even though a client has called the **OleLockServer** function.

See Also **OleUnlockServer**

OleObjectConvert 3.1

Syntax

```
#include <ole.h>
OLESTATUS OleObjectConvert(lpObject, lpszProtocol, lpClient,
lhClientDoc, lpszObjname, lplpObject)
```

```
function OleObjectConvert(OleObject: POleObject; Protocol: PChar;
Client: POleClient; ClientDoc: LHClientDoc; ObjName: PChar; var
OleObject: POleObject): TOleStatus;
```

The **OleObjectConvert** function creates a new object that supports a specified protocol by converting an existing object. This function neither deletes nor replaces the original object.

Parameters

lpObject	Points to the object to convert.
lpszProtocol	Points to a null-terminated string specifying the name of the required protocol. Currently this value can be Static (for uneditable pictures only).
lpClient	Points to an **OLECLIENT** structure for the new object.
lhClientDoc	Identifies the client document in which the object is being created.
lpszObjname	Points to a null-terminated string specifying the client's name for the object. This name must be unique with respect to the names of any other objects in the document and cannot contain a slash mark (/).
lplpObject	Points to a variable in which the library stores a pointer to the new object.

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_BUSY
OLE_ERROR_HANDLE
OLE_ERROR_NAME
OLE_ERROR_OBJECT
OLE_ERROR_STATIC

Comments The only conversion currently supported is that of changing a linked or embedded object to a static object.

See Also **OleClone**

OleQueryBounds 3.1

Syntax #include <ole.h>
OLESTATUS OleQueryBounds(lpObject, lpBounds)

function OleQueryBounds(Self: POleObject; var Bounds: TRect): TOleStatus;

The **OleQueryBounds** function retrieves the extents of the bounding rectangle on the target device for the specified object. The coordinates are in MM_HIMETRIC units.

Parameters

lpObject Points to the object to query.

lpBounds Points to a **RECT** structure for the extents of the bounding rectangle. The members of the **RECT** structure have the following meanings:

Member	Meaning
rect.left	0
rect.top	0
rect.right	x-extent
rect.bottom	y-extent

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_MEMORY
OLE_ERROR_OBJECT

See Also **OleSetBounds, SetMapMode**

OleQueryClientVersion 3.1

Syntax
```
#include <ole.h>
DWORD OleQueryClientVersion(void)
```

```
function OleQueryClientVersion: Longint;
```

The **OleQueryClientVersion** function retrieves the version number of the client library.

Parameters This function has no parameters.

Return Value The return value is a doubleword value. The major version number is in the low-order byte of the low-order word, and the minor version number is in the high-order byte of the low-order word. The high-order word is reserved.

See Also **OleQueryServerVersion**

OleQueryCreateFromClip 3.1

Syntax
```
#include <ole.h>
OLESTATUS OleQueryCreateFromClip(lpszProtocol, renderopt, cfFormat)
```

```
function OleQueryCreateFromClip(Protocol: PChar; render_opt: TOleOPT_Render; Format: TOleClipFormat): TOleStatus;
```

The **OleQueryCreateFromClip** function checks whether the object on the clipboard supports the specified protocol and rendering options.

Parameters

lpszProtocol Points to a null-terminated string specifying the name of the protocol needed by the client. Currently, this value can be StdFileEditing (the name of the object linking and embedding protocol) or Static (for uneditable pictures only).

renderopt Specifies the client's preference for presentation data for the object. This parameter can be one of the following values:

Value	Meaning
olerender_draw	The client calls the **OleDraw** function, and the library obtains and manages presentation data.
olerender_format	The library obtains and manages the data in the requested format, as specified by the *cfFormat* parameter.
olerender_none	The client library does not obtain any presentation data and does not draw the object.

cfFormat Specifies the clipboard format. This parameter is used only when the *renderopt* parameter is **olerender_format**. If the clipboard format is CF_METAFILEPICT, CF_DIB, or CF_BITMAP, the library manages the data and draws the object. The library does not support drawing for any other formats.

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_FORMAT
OLE_ERROR_PROTOCOL

Comments The **OleQueryCreateFromClip** function is typically used to check whether to enable a Paste command.

The **olerender_none** rendering option is typically used to support hyperlinks. With this option, the client does not call **OleDraw** and calls the **OleGetData** function only for ObjectLink, OwnerLink, and Native formats.

The **olerender_format** rendering option allows a client to compute data (instead of painting it), use an unusual data format, or modify a standard data format. With this option the client does not call **OleDraw**. The client calls **OleGetData** to retrieve data in the specified format.

The **olerender_draw** rendering option is the most typical option. It is the easiest rendering option for the client to implement (the client simply calls **OleDraw**), and it allows the most flexibility. An object handler can exploit this flexibility to store no presentation data, a private presentation data format, or several different formats that it can choose among dynamically. Future implementations of object linking and embedding (OLE) may also exploit the flexibility that is inherent in this option.

See Also **OleCreateFromClip, OleDraw, OleGetData**

OleQueryLinkFromClip 3.1

Syntax #include <ole.h>
OLESTATUS OleQueryLinkFromClip(lpszProtocol, renderopt, cfFormat)

function OleQueryLinkFromClip(Protocol: PChar; render_opt: TOleOPT_Render; Format: TOleClipFormat): TOleStatus;

The **OleQueryLinkFromClip** function checks whether a client application can use the data on the clipboard to produce a linked object that supports the specified protocol and rendering options.

Parameters

lpszProtocol Points to a null-terminated string specifying the name of the protocol needed by the client. Currently this value can be StdFileEditing (the name of the object linking and embedding protocol).

renderopt Specifies the client's preference for presentation data for the object. This parameter can be one of the following values:

Value	Meaning
olerender_draw	The client calls the **OleDraw** function, and the library obtains and manages presentation data.
olerender_format	The library obtains and manages the data in the requested format, as specified by the *cfFormat* parameter.
olerender_none	The client library does not obtain any presentation data and does not draw the object.

cfFormat Specifies the clipboard format. This parameter is used only when the *renderopt* parameter is **olerender_format**. If this clipboard format is CF_METAFILEPICT, CF_DIB, or CF_BITMAP, the library manages the data and draws the object. The library does not support drawing for any other formats.

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_FORMAT
OLE_ERROR_PROTOCOL

Comments The **OleQueryLinkFromClip** function is typically used to check whether to enable a Paste Link command.

The **olerender_none** rendering option is typically used to support hyperlinks. With this option, the client does not call **OleDraw** and calls the **OleGetData** function only for ObjectLink, OwnerLink, and Native formats.

The **olerender_format** rendering option allows a client to compute data (instead of painting it), use an unusual data format, or modify a standard data format. With this option, the client does not call **OleDraw**. The client calls **OleGetData** to retrieve data in the specified format.

The **olerender_draw** rendering option is the most typical option. It is the easiest rendering option for the client to implement (the client simply calls **OleDraw**), and it allows the most flexibility. An object handler can exploit this flexibility to store no presentation data, a private presentation data format, or several different formats that it can choose among dynamically. Future implementations of object linking and embedding (OLE) may also exploit the flexibility that is inherent in this option.

See Also **OleCreateLinkFromClip, OleDraw, OleGetData**

OleQueryName 3.1

Syntax

```
#include <ole.h>
OLESTATUS OleQueryName(lpObject, lpszObject, lpwBuffSize)
```

```
function OleQueryName(Self: POleObject; Name: PChar; var NameSize:
Word): TOleStatus;
```

The **OleQueryName** function retrieves the name of a specified object.

Parameters

lpObject	Points to the object whose name is being queried.
lpszObject	Points to a character array that contains a null-terminated string. When the function returns, this string specifies the name of the object.
lpwBuffSize	Points to a variable containing the size, in bytes, of the buffer pointed to by the *lpszObject* parameter. When the function returns, this value is the number of bytes copied to the buffer.

Return Value The return value is OLE_OK if the function is successful. Otherwise, it is an error value, which may be OLE_ERROR_OBJECT.

See Also **OleRename**

OleQueryOpen 3.1

Syntax #include <ole.h>
OLESTATUS OleQueryOpen(lpObject)

function OleQueryOpen(Self: POleObject): TOleStatus;

The **OleQueryOpen** function checks whether the specified object is open.

Parameters *lpObject* Points to the object to query.

Return Value The return value is OLE_OK if the object is open. Otherwise, it is an error value, which may be one of the following:

OLE_ERROR_COMM
OLE_ERROR_OBJECT
OLE_ERROR_STATIC

See Also **OleActivate**

OleQueryOutOfDate 3.1

Syntax #include <ole.h>
OLESTATUS OleQueryOutOfDate(lpObject)

function OleQueryOutOfDate(Self: POleObject): TOleStatus;

The **OleQueryOutOfDate** function checks whether an object is out-of-date.

Parameters *lpObject* Points to the object to query.

Return Value The return value is OLE_OK if the object is up-to-date. Otherwise, it is an error value, which may be one of the following:

OLE_ERROR_OBJECT
OLE_ERROR_OUTOFDATE

Comments The **OleQueryOutOfDate** function has not been implemented for the current version of object linking and embedding (OLE). For linked objects, **OleQueryOutOfDate** always returns OLE_OK.

A linked object might be out-of-date if the document that is the source for the link has been updated. An embedded object that contains links to other objects might also be out-of-date.

See Also **OleEqual, OleUpdate**

OleQueryProtocol 3.1

Syntax #include <ole.h>
void FAR* OleQueryProtocol(lpobj, lpszProtocol)

function OleQueryProtocol(Self: POleObject; Protocol: PChar): Pointer;

The **OleQueryProtocol** function checks whether an object supports a specified protocol.

Parameters

lpobj	Points to the object to query.
lpszProtocol	Points to a null-terminated string specifying the name of the requested protocol. This value can be StdFileEditing or StdExecute.

Return Value The return value is a void pointer to an **OLEOBJECT** structure if the function is successful, or it is NULL if the object does not support the requested protocol. The library can return OLE_WAIT_FOR_RELEASE when an application calls this function.

Comments The **OleQueryProtocol** function queries whether the specified protocol is supported and returns a modified object pointer that allows access to the function table for the protocol. This modified object pointer points to a structure that has the same form as the **OLEOBJECT** structure; the new structure also points to a table of functions and may contain additional state information. The new pointer does not point to a different object—if the object is deleted, secondary pointers become invalid. If a protocol includes delete functions, calling a delete function invalidates all pointers to that object.

A client application typically calls **OleQueryProtocol**, specifying StdExecute for the *lpszProtocol* parameter, before calling the **OleExecute** function. This allows the client application to check whether the server for an object supports dynamic data exchange (DDE) execute commands.

See Also **OleExecute**

OleQueryReleaseError 3.1

Syntax

```
#include <ole.h>
OLESTATUS OleQueryReleaseError(lpobj)
```

```
function OleQueryReleaseError(Self: POleObject): TOleStatus;
```

The **OleQueryReleaseError** function checks the error value for an asynchronous operation on an object.

Parameters

lpobj	Points to an object for which the error value is to be queried.

Return Value The return value, if the function is successful, is either OLE_OK if the asynchronous operation completed successfully or the error value for that operation. If the pointer passed in the *lpobj* parameter is invalid, the function returns OLE_ERROR_OBJECT.

Comments A client application receives the OLE_RELEASE notification when an asynchronous operation has terminated. The client should then call **OleQueryReleaseError** to check whether the operation has terminated successfully or with an error value.

See Also **OleQueryReleaseMethod, OleQueryReleaseStatus**

OleQueryReleaseMethod 3.1

Syntax

```
#include <ole.h>
OLE_RELEASE_METHOD OleQueryReleaseMethod(lpobj)
```

```
function OleQueryReleaseMethod(Self: POleObject):
TOle_Release_Method;
```

The **OleQueryReleaseMethod** function finds out the operation that finished for the specified object.

Parameters *lpobj* Points to an object for which the operation is to be queried.

Return Value The return value indicates the server operation (method) that finished. It can be one of the following values:

Value	Server operation
OLE_ACTIVATE	Activate
OLE_CLOSE	Close
OLE_COPYFROMLNK	CopyFromLink (autoreconnect)
OLE_CREATE	Create
OLE_CREATEFROMFILE	CreateFromFile
OLE_CREATEFROMTEMPLATE	CreateFromTemplate
OLE_CREATEINVISIBLE	CreateInvisible
OLE_CREATELINKFROMFILE	CreateLinkFromFile
OLE_DELETE	Object Delete
OLE_EMBPASTE	Paste and Update
OLE_LNKPASTE	PasteLink (autoreconnect)
OLE_LOADFROMSTREAM	LoadFromStream (autoreconnect)
OLE_NONE	No operation active
OLE_OTHER	Other miscellaneous asynchronous operations
OLE_RECONNECT	Reconnect
OLE_REQUESTDATA	OleRequestData
OLE_RUN	Run
OLE_SERVERUNLAUNCH	Server is stopping
OLE_SETDATA	OleSetData
OLE_SETUPDATEOPTIONS	Setting update options
OLE_SHOW	Show
OLE_UPDATE	Update

If the pointer passed in the *lpobj* parameter is invalid, the function returns OLE_ERROR_OBJECT.

Comments A client application receives the OLE_RELEASE notification when an asynchronous operation has ended. The client can then call **OleQueryReleaseMethod** to check which operation caused the library to send the OLE_RELEASE notification. The client calls **OleQueryReleaseError** to determine whether the operation terminated successfully or with an error value.

See Also **OleQueryReleaseError, OleQueryReleaseStatus**

OleQueryReleaseStatus 3.1

Syntax #include <ole.h>
OLESTATUS OleQueryReleaseStatus(lpobj)

function OleQueryReleaseStatus(Self: POleObject): TOleStatus;

The **OleQueryReleaseStatus** function determines whether an operation has finished for the specified object.

Parameters *lpobj* Points to an object for which the operation is queried.

Return Value The return value, if the function is successful, is either OLE_BUSY if an operation is in progress or OLE_OK. If the pointer passed in the *lpobj* parameter is invalid, the function returns OLE_ERROR_OBJECT.

See Also **OleQueryReleaseError, OleQueryReleaseMethod**

OleQueryServerVersion 3.1

Syntax #include <ole.h>
DWORD OleQueryServerVersion(void)

function OleQueryServerVersion: Longint;

The **OleQueryServerVersion** function retrieves the version number of the server library.

Parameters This function has no parameters.

Return Value The return value is a doubleword value. The major version number is in the low-order byte of the low-order word, and the minor version number is in the high-order byte of the low-order word. The high-order word is reserved.

See Also **OleQueryClientVersion**

OleQuerySize 3.1

Syntax #include <ole.h>
OLESTATUS OleQuerySize(lpObject, pdwSize)

function OleQuerySize(Self: POleObject; var Size: Longint): TOleStatus;

The **OleQuerySize** function retrieves the size of the specified object.

Parameters

lpObject Points to the object to query.

pdwSize Points to a variable for the size of the object. This variable contains the size of the object when the function returns.

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_MEMORY
OLE_ERROR_OBJECT

See Also **OleLoadFromStream**

OleQueryType 3.1

Syntax #include <ole.h>
OLESTATUS OleQueryType(lpObject, lpType)

function OleQueryType(Self: POleObject; var LinkType: Longint): TOleStatus;

The **OleQueryType** function checks whether a specified object is embedded, linked, or static.

Parameters

lpObject Points to the object for which the type is to be queried.

lpType Points to a long variable that contains the type of the object when the function returns. This parameter can be one of the following values:

Value	Meaning
OT_EMBEDDED	Object is embedded.
OT_LINK	Object is a link.
OT_STATIC	Object is a static picture.

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_GENERIC
OLE_ERROR_OBJECT

See Also **OleEnumFormats**

OleReconnect 3.1

Syntax

```
#include <ole.h>
OLESTATUS OleReconnect(lpObject)

function OleReconnect(Self: POleObject): TOleStatus;
```

The **OleReconnect** function reestablishes a link to an open linked object. If the specified object is not open, this function does not open it.

Parameters

lpObject	Points to the object to reconnect to.

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_BUSY
OLE_ERROR_NOT_LINK
OLE_ERROR_OBJECT
OLE_ERROR_STATIC
OLE_WAIT_FOR_RELEASE

Comments A client application can use **OleReconnect** to keep the presentation for a linked object up-to-date.

See Also **OleActivate, OleClose, OleUpdate**

OleRegisterClientDoc 3.1

Syntax

```
#include <ole.h>
OLESTATUS OleRegisterClientDoc(lpszClass, lpszDoc, reserved, lplhDoc)

function OleRegisterClientDoc(Class, Doc: PChar; Reserved: Longint; var
Doc: LHClientDoc): TOleStatus;
```

The **OleRegisterClientDoc** function registers an open client document with the library and returns the handle of that document.

Parameters

lpszClass	Points to a null-terminated string specifying the class of the client document.
lpszDoc	Points to a null-terminated string specifying the location of the client document. (This value should be a fully qualified path.)
reserved	Reserved. Must be zero.
lplhDoc	Points to the handle of the client document when the function returns. This handle is used to identify the document in other document-management functions.

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_ALREADY_REGISTERED
OLE_ERROR_MEMORY
OLE_ERROR_NAME

Comments When a document being copied onto the clipboard exists only because the client application is copying Native data that contains objects, the name specified in the *lpszDoc* parameter must be Clipboard.

Client applications should register open documents with the library and notify the library when a document is renamed, closed, saved, or restored to a changed state.

See Also **OleRenameClientDoc**, **OleRevertClientDoc**, **OleRevokeClientDoc**, **OleSavedClientDoc**

OleRegisterServer 3.1

Syntax #include <ole.h>
OLESTATUS OleRegisterServer(lpszClass, lpsrvr, lplhserver, hinst, srvruse)

function OleRegisterServer(Class: PChar; ServerDef: POleServer; var Server: LHServer; Inst: THandle; Use: TOle_Server_Use): TOleStatus;

The **OleRegisterServer** function registers the specified server, class name, and instance with the server library.

Parameters

lpszClass	Points to a null-terminated string specifying the class name being registered.
lpsrvr	Points to an **OLESERVER** structure allocated and initialized by the server application.
lplhserver	Points to a variable of type **LHSERVER** in which the library stores the handle of the server. This handle is used in such functions as **OleRegisterServerDoc** and **OleRevokeServer**.
hinst	Identifies the instance of the server application. This handle is used to ensure that clients connect to the correct instance of a server application.
srvruse	Specifies whether the server uses a single instance or multiple instances to support multiple objects. This value must be either OLE_SERVER_SINGLE or OLE_SERVER_MULTI.

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_CLASS
OLE_ERROR_MEMORY
OLE_ERROR_PROTECT_ONLY

Comments

When the server application starts, it creates an **OLESERVER** structure and calls the **OleRegisterServer** function. Servers that support several class names can allocate a structure for each or reuse the same structure. The class name is passed to server-application functions that are called through the library, so that servers supporting more than one class can check which class is being requested.

The *srvruse* parameter is used when the libraries open an object. When OLE_SERVER_MULTI is specified for this parameter and all current instances are already editing an object, a new instance of the server is started. Servers that support the multiple document interface (MDI) typically specify OLE_SERVER_SINGLE.

See Also

OleRegisterServerDoc, OleRevokeServer

OleRegisterServerDoc 3.1

Syntax #include <ole.h>
OLESTATUS OleRegisterServerDoc(lhsrvr, lpszDocName, lpdoc, lplhdoc)

function OleRegisterServerDoc(Server: LHServer; DocName: PChar; DocDef: POleServerDoc; var Doc: LHServerDoc): TOleStatus;

The **OleRegisterServerDoc** function registers a document with the server library in case other client applications have links to it. A server application uses this function when the server is started with the **/Embedding** *filename* option or when it creates or opens a document that is not requested by the library.

Parameters

lhsrvr	Identifies the server. Server applications obtain this handle by calling the **OleRegisterServer** function.
lpszDocName	Points to a null-terminated string specifying the permanent name for the document. This parameter should be a fully qualified path.
lpdoc	Points to an **OLESERVERDOC** structure allocated and initialized by the server application.
lplhdoc	Points to a handle that will identify the document. This parameter points to the handle when the function returns.

Return Value If the function is successful, the return value is OLE_OK. Otherwise, it is an error value, which may be one of the following:

OLE_ERROR_ADDRESS
OLE_ERROR_HANDLE
OLE_ERROR_MEMORY

Comments If the document was created or opened in response to a request from the server library, the server should not register the document by using **OleRegisterServerDoc.** Instead, the server should return a pointer to the **OLESERVERDOC** structure through the parameter to the relevant function.

See Also **OleRegisterServer, OleRevokeServerDoc**

OleRelease 3.1

Syntax #include <ole.h>
OLESTATUS OleRelease(lpObject)

function OleRelease(Self: POleObject): TOleStatus;

The **OleRelease** function releases an object from memory and closes it if it was open. This function does not indicate that the object has been deleted from the client document.

Parameters

lpObject	Points to the object to release.

Return Value If the function is successful, the return value is OLE_OK. Otherwise, it is an error value, which may be one of the following:

OLE_BUSY
OLE_ERROR_OBJECT
OLE_WAIT_FOR_RELEASE

Comments The **OleRelease** function should be called for all objects when closing the client document.

See Also **OleDelete**

OleRename 3.1

Syntax #include <ole.h>
OLESTATUS OleRename(lpObject, lpszNewname)

function OleRename(Self: POleObject; NewName: PChar): TOleStatus;

The **OleRename** function renames an object.

Parameters

lpObject	Points to the object that is being renamed.
lpszNewname	Points to a null-terminated string specifying the new name of the object.

Return Value The return value is OLE_OK if the function is successful. Otherwise, it is an error value, which may be OLE_ERROR_OBJECT.

Comments Object names need not be seen by the user. They must be unique within the containing document and must be preserved when the document is saved.

See Also **OleQueryName**

OleRenameClientDoc 3.1

Syntax #include <ole.h>
OLESTATUS OleRenameClientDoc(lhClientDoc, lpszNewDocname)

function OleRenameClientDoc(ClientDoc: LHClientDoc; NewDocName; PChar): TOleStatus;

The **OleRenameClientDoc** function informs the client library that a document has been renamed. A client application calls this function when a document name has changed—for example, when the user chooses the Save or Save As command from the File menu.

Parameters

lhClientDoc	Identifies the document that has been renamed.
lpszNewDocname	Points to a null-terminated string specifying the new name of the document.

Return Value The return value is OLE_OK if the function is successful. Otherwise, it is an error value, which may be OLE_ERROR_HANDLE.

Comments Client applications should register open documents with the library and notify the library when a document is renamed, closed, saved, or restored to a changed state.

See Also **OleRegisterClientDoc, OleRevertClientDoc, OleRevokeClientDoc, OleSavedClientDoc**

OleRenameServerDoc 3.1

Syntax #include <ole.h>
OLESTATUS OleRenameServerDoc(lhDoc, lpszDocName)

function OleRenameServerDoc(Doc: LHServerDoc; NewName: PChar): TOleStatus;

The **OleRenameServerDoc** function informs the server library that a document has been renamed.

Parameters

lhDoc	Identifies the document that has been renamed.
lpszDocName	Points to a null-terminated string specifying the new name of the document. This parameter is typically a fully qualified path.

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_HANDLE
OLE_ERROR_MEMORY

Comments The **OleRenameServerDoc** function has the same effect as sending the OLE_RENAMED notification to the client application's callback function. The server application calls this function when it renames a document to which the active links need to be reconnected or when the user chooses the Save As command from the File menu while working with an embedded object.

Server applications should register open documents with the server library and notify the library when a document is renamed, closed, saved, or restored to a changed state.

See Also **OleRegisterServerDoc, OleRevertServerDoc, OleRevokeServerDoc, OleSavedServerDoc**

OleRequestData 3.1

Syntax #include <ole.h>
OLESTATUS OleRequestData(lpObject, cfFormat)

function OleRequestData(Self: POleObject; Format: TOleClipFormat): TOleStatus;

The **OleRequestData** function requests the library to retrieve data in a specified format from a server.

Parameters

lpObject	Points to the object that is associated with the server from which data is to be retrieved.
cfFormat	Specifies the format in which data is to be returned. This parameter can be one of the predefined clipboard formats or the value returned by the **RegisterClipboardFormat** function.

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_BUSY
OLE_ERROR_NOT_OPEN
OLE_ERROR_OBJECT
OLE_ERROR_STATIC
OLE_WAIT_FOR_RELEASE

Comments The client application should be connected to the server application when the client calls the **OleRequestData** function. When the client receives the OLE_RELEASE notification, it can retrieve the data from the object by using the **OleGetData** function or query the data by using such functions as **OleQueryBounds**.

If the requested data format is the same as the presentation data for the object, the library manages the data and updates the presentation.

The **OleRequestData** function returns OLE_WAIT_FOR_RELEASE if the server is busy. In this case, the application should continue to dispatch messages until it receives a callback notification with the OLE_RELEASE argument.

See Also **OleEnumFormats, OleGetData, OleSetData, RegisterClipboardFormat**

OleRevertClientDoc 3.1

Syntax

```
#include <ole.h>
OLESTATUS OleRevertClientDoc(lhClientDoc)

function OleRevertClientDoc(ClientDoc: LHClientDoc): TOleStatus;
```

The **OleRevertClientDoc** function informs the library that a document has been restored to a previously saved condition.

Parameters *lhClientDoc* Identifies the document that has been restored to its saved state.

Return Value The return value is OLE_OK if the function is successful. Otherwise, it is an error value, which may be OLE_ERROR_HANDLE.

Comments A client application should call the **OleRevertClientDoc** function when it reloads a document without saving changes to the document.

Client applications should register open documents with the library and notify the library when a document is renamed, closed, saved, or restored to a saved state.

See Also **OleRegisterClientDoc, OleRenameClientDoc, OleRevokeClientDoc, OleSavedClientDoc**

OleRevertServerDoc 3.1

Syntax #include <ole.h>
OLESTATUS OleRevertServerDoc(lhDoc)

function OleRevertServerDoc(Doc: LHServerDoc): TOleStatus;

The **OleRevertServerDoc** function informs the server library that the server has restored a document to its saved state without closing it.

Parameters *lhDoc* Identifies the document that has been restored to its saved state.

Return Value The return value is OLE_OK if the function is successful. Otherwise, it is an error value, which may be OLE_ERROR_HANDLE.

Comments Server applications should register open documents with the server library and notify the library when a document is renamed, closed, saved, or restored to a saved state.

See Also **OleRegisterServerDoc, OleRenameServerDoc, OleRevokeServerDoc, OleSavedServerDoc**

OleRevokeClientDoc 3.1

Syntax #include <ole.h>
OLESTATUS OleRevokeClientDoc(lhClientDoc)

function OleRevokeClientDoc(ClientDoc: LHClientDoc): TOleStatus;

The **OleRevokeClientDoc** function informs the client library that a document is no longer open.

Parameters *lhClientDoc* Identifies the document that is no longer open. This handle is invalid following the call to **OleRevokeClientDoc**.

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_HANDLE
OLE_ERROR_NOT_EMPTY

Comments The client application should delete all the objects in a document before calling **OleRevokeClientDoc**.

Client applications should register open documents with the library and notify the library when a document is renamed, closed, saved, or restored to a changed state.

See Also **OleRegisterClientDoc, OleRenameClientDoc, OleRevertClientDoc, OleSavedClientDoc**

OleRevokeObject 3.1

Syntax #include <ole.h>
OLESTATUS OleRevokeObject(lpClient)

function OleRevokeObject(Client: POleClient): TOleStatus;

The **OleRevokeObject** function revokes access to an object. A server application typically calls this function when the user destroys an object.

Parameters *lpClient* Points to the **OLECLIENT** structure associated with the object being revoked.

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

See Also **OleRevokeServer, OleRevokeServerDoc**

OleRevokeServer 3.1

Syntax #include <ole.h>
OLESTATUS OleRevokeServer(lhServer)

function OleRevokeServer(Server: LHServer): TOleStatus;

The **OleRevokeServer** function is called by a server application to close any registered documents.

Parameters *lhServer* Identifies the server to revoke. A server application obtains this handle in a call to the **OleRegisterServer** function.

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_HANDLE
OLE_WAIT_FOR_RELEASE

Comments The **OleRevokeServer** function returns OLE_WAIT_FOR_RELEASE if communications between clients and the server are in the process of terminating. In this case, the server application should continue to send and dispatch messages until the library calls the server's **Release** function.

See Also **OleRegisterServer, OleRevokeObject, OleRevokeServerDoc**

OleRevokeServerDoc 3.1

Syntax #include <ole.h>
OLESTATUS OleRevokeServerDoc(lhdoc)

function OleRevokeServerDoc(Doc: LHServerDoc): TOleStatus;

The **OleRevokeServerDoc** function revokes the specified document. A server application calls this function when a registered document is being closed or otherwise made unavailable to client applications.

Parameters *lhdoc* Identifies the document to revoke. This handle was returned by a call to the **OleRegisterServerDoc** function or was associated with a document by using one of the server-supplied functions that create documents.

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_HANDLE
OLE_WAIT_FOR_RELEASE

Comments If this function returns OLE_WAIT_FOR_RELEASE, the server application should not free the **OLESERVERDOC** structure or exit until the library calls the server's **Release** function.

See Also **OleRegisterServerDoc, OleRevokeObject, OleRevokeServer**

OleSavedClientDoc 3.1

Syntax #include <ole.h>
OLESTATUS OleSavedClientDoc(lhClientDoc)

function OleSavedClientDoc(ClientDoc: LHClientDoc): TOleStatus;

The **OleSavedClientDoc** function informs the client library that a document has been saved.

Parameters *lhClientDoc* Identifies the document that has been saved.

Return Value The return value is OLE_OK if the function is successful. Otherwise, it is an error value, which may be OLE_ERROR_HANDLE.

Comments Client applications should register open documents with the client library and notify the library when a document is renamed, closed, saved, or restored to a saved state.

See Also **OleRegisterClientDoc, OleRenameClientDoc, OleRevertClientDoc, OleRevokeClientDoc**

OleSavedServerDoc 3.1

Syntax #include <ole.h>
OLESTATUS OleSavedServerDoc(lhDoc)

function OleSavedServerDoc(Doc: LHServerDoc): TOleStatus;

The **OleSavedServerDoc** function informs the server library that a document has been saved.

Parameters *lhDoc* Identifies the document that has been saved.

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_CANT_UPDATE_CLIENT
OLE_ERROR_HANDLE

Comments The **OleSavedServerDoc** function has the same effect as sending the OLE_SAVED notification to the client application's callback function. The server application calls this function when saving a document or when updating an embedded object without closing the document.

When a server application receives the OLE_ERROR_CANT_UPDATE_CLIENT error value, it should display a message box indicating that the user cannot update the document until the server terminates.

Server applications should register open documents with the server library and notify the library when a document is renamed, closed, saved, or restored to a saved state.

See Also **OleRegisterServerDoc, OleRenameServerDoc, OleRevertServerDoc, OleRevokeServerDoc**

OleSaveToStream 3.1

Syntax #include <ole.h>
OLESTATUS OleSaveToStream(lpObject, lpStream)

function OleSaveToStream(Self: POleObject; Stream: POleStream): TOleStatus;

The **OleSaveToStream** function saves an object to the stream.

Parameters

lpObject Points to the object to be saved to the stream.

lpStream Points to an **OLESTREAM** structure allocated and initialized by the client application. The library calls the **Put** function in the **OLESTREAM** structure to store the data from the object.

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_MEMORY
OLE_ERROR_OBJECT
OLE_ERROR_STREAM

Comments An application can use the **OleQuerySize** function to find the number of bytes to allocate for the object.

See Also **OleLoadFromStream, OleQuerySize**

OleSetBounds 3.1

Syntax

```
#include <ole.h>
OLESTATUS OleSetBounds(lpObject, lprcBound)
```

```
function OleSetBounds(Self: POleObject; var Bounds: TRect): TOleStatus;
```

The **OleSetBounds** function sets the coordinates of the bounding rectangle for the specified object on the target device.

Parameters

lpObject Points to the object for which the bounding rectangle is set.

lprcBound Points to a **RECT** structure containing the coordinates of the bounding rectangle. The coordinates are specified in MM_HIMETRIC units. Neither the width nor height of an object should exceed 32,767 MM_HIMETRIC units.

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_BUSY
OLE_ERROR_MEMORY
OLE_ERROR_OBJECT
OLE_WAIT_FOR_RELEASE

The **OleSetBounds** function returns OLE_ERROR_OBJECT when it is called for a linked object.

Comments The **OleSetBounds** function is ignored for linked objects, because the size of a linked object is determined by the source document for the link.

A client application uses **OleSetBounds** to change the bounding rectangle. The client does not need to call **OleSetBounds** every time a server is opened.

The bounding rectangle specified in the **OleSetBounds** function does not necessarily have the same dimensions as the rectangle specified in the call to the **OleDraw** function. These dimensions may be different because of the view scaling used by the container application. An application can use **OleSetBounds** to cause the server to reformat the picture to fit the rectangle more closely.

In the MM_HIMETRIC mapping mode, the positive y-direction is up.

See Also **OleDraw, OleQueryBounds, SetMapMode**

OleSetColorScheme 3.1

Syntax

```
#include <ole.h>
OLESTATUS OleSetColorScheme(lpObject, lpPalette)

function OleSetColorScheme(Self: POleObject; var Palette: TLogPalette):
TOleStatus;
```

The **OleSetColorScheme** function specifies the palette a client application recommends be used when the server application edits the specified object. The server application can ignore the recommended palette.

Parameters

lpObject	Points to an **OLEOBJECT** structure describing the object for which a palette is recommended.
lpPalette	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, which may be one of the following:

OLE_BUSY
OLE_ERROR_COMM
OLE_ERROR_MEMORY

OLE_ERROR_OBJECT
OLE_ERROR_PALETTE
OLE_ERROR_STATIC
OLE_WAIT_FOR_RELEASE

The **OleSetColorScheme** function returns OLE_ERROR_OBJECT when it is called for a linked object.

Comments A client application uses **OleSetColorScheme** to change the color scheme. The client does not need to call **OleSetColorScheme** every time a server is opened.

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 should specify an even number of palette entries. When there is an uneven number of entries, the server interprets the odd entry as a fill color; that is, if there are five entries, three are interpreted as fill colors and two as line and text colors.

When server applications render metafiles, they should use the suggested palette.

OleSetData 3.1

Syntax #include <ole.h>
OLESTATUS OleSetData(lpObject, cfFormat, hData)

function OleSetData(Self: POleObject; Format: TOleClipFormat; Data: THandle): TOleStatus;

The **OleSetData** function sends data in the specified format to the server associated with a specified object.

Parameters

lpObject	Points to an object specifying the server to which data is to be sent.
cfFormat	Specifies the format of the data.
hData	Identifies a memory object containing the data in the specified format.

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_BUSY
OLE_ERROR_BLANK
OLE_ERROR_MEMORY
OLE_ERROR_NOT_OPEN
OLE_ERROR_OBJECT
OLE_WAIT_FOR_RELEASE

If the specified object cannot accept the data, the function returns an error value. If the server is not open and the requested data format is different from the format of the presentation data, the return value is OLE_ERROR_NOT_OPEN.

See Also **OleGetData, OleRequestData**

OleSetHostNames 3.1

Syntax #include <ole.h>
OLESTATUS OleSetHostNames(lpObject, lpszClient, lpszClientObj)

function OleSetHostNames(Self: POleObject; ClientName, ObjectName: PChar): TOleStatus;

The **OleSetHostNames** function specifies the name of the client application and the client's name for the specified object. This information is used in window titles when the object is being edited in the server application.

Parameters

lpObject	Points to the object for which a name is to be set.
lpszClient	Points to a null-terminated string specifying the name of the client application.
lpszClientObj	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, which may be one of the following:

OLE_BUSY
OLE_ERROR_MEMORY
OLE_ERROR_OBJECT
OLE_WAIT_FOR_RELEASE

The **OleSetHostNames** function returns OLE_ERROR_OBJECT when it is called for a linked object.

Comments When a server application is started for editing of an embedded object, it displays in its title bar the string specified in the *lpszClientObj* parameter. The object name specified in this string should be the name of the client document containing the object.

A client application uses **OleSetHostNames** to set the name of an object the first time that object is activated or to change the name of an object. The client does not need to call **OleSetHostNames** every time a server is opened.

OleSetLinkUpdateOptions 3.1

Syntax

```
#include <ole.h>
OLESTATUS OleSetLinkUpdateOptions(lpObject, UpdateOpt)
```

```
function OleSetLinkUpdateOptions(Self: POleObject; UpdateOpt:
TOleOpt_Update): TOleStatus;
```

The **OleSetLinkUpdateOptions** function sets the link-update options for the presentation of the specified object.

Parameters

lpObject Points to the object for which the link-update option is set.

UpdateOpt Specifies the link-update option for the specified object. This parameter can be one of the following values:

Option	Description
oleupdate_always	Update the linked object whenever possible. This option supports the Automatic link-update radio button in the Links dialog box.
oleupdate_oncall	Update the linked object only on request from the client application. This option supports the Manual link-update radio button in the Links dialog box.
oleupdate_onsave	Update the linked object when the source document is saved by the server.

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_BUSY
OLE_ERROR_OBJECT
OLE_ERROR_OPTION
OLE_ERROR_STATIC
OLE_WAIT_FOR_RELEASE

See Also **OleGetLinkUpdateOptions**

OleSetTargetDevice 3.1

Syntax

```
#include <ole.h>
OLESTATUS OleSetTargetDevice(lpObject, hotd)

function OleSetTargetDevice(Self: POleObject; TargetDevice: THandle):
TOleStatus;
```

The **OleSetTargetDevice** function specifies the target output device for an object.

Parameters

lpObject	Points to the object for which a target device is specified.
hotd	Identifies an **OLETARGETDEVICE** structure that describes the target device for the object.

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_BUSY
OLE_ERROR_MEMORY
OLE_ERROR_OBJECT
OLE_ERROR_STATIC
OLE_WAIT_FOR_RELEASE

Comments The **OleSetTargetDevice** function allows a linked or embedded object to be formatted correctly for a target device, even when the object is rendered on a different device. A client application should call this function whenever the target device changes, so that servers can be notified to change the rendering of the object, if necessary. The client application should call the **OleUpdate** function to ensure that the information is sent to the server, so that the server can make the necessary changes to the object's presentation. The client application should call the library to redraw the object if it receives a notification from the server that the object has changed.

A client application uses the **OleSetTargetDevice** function to change the target device. The client does not need to call **OleSetTargetDevice** every time a server is opened.

OleUnblockServer 3.1

Syntax

#include <ole.h>
OLESTATUS OleUnblockServer(lhSrvr, lpfRequest)

function OleUnblockServer(Server: LHServer; var Requests: Bool): TOleStatus;

The **OleUnblockServer** function processes a request from a queue created by calling the **OleBlockServer** function.

Parameters

lhSrvr	Identifies the server for which requests were queued.
lpfRequest	Points to a flag indicating whether there are further requests in the queue. If there are further requests in the queue, this flag is TRUE when the function returns. Otherwise, it is FALSE when the function returns.

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_HANDLE
OLE_ERROR_MEMORY

Comments

A server application can use the **OleBlockServer** and **OleUnblockServer** functions to control when the server library processes requests from client applications. It is best to use **OleUnblockServer** outside the **GetMessage** function in a message loop, unblocking all blocked messages before getting the next message. Unblocking message loops should not be run inside server-defined functions that are called by the library.

See Also

OleBlockServer

OleUnlockServer 3.1

Syntax #include <ole.h>
OLESTATUS OleUnlockServer(hServer)

function OleUnlockServer(Server: LHServer): TOleStatus;

The **OleUnlockServer** function unlocks a server that was locked by the **OleLockServer** function.

Parameters

hServer	Identifies the server to release from memory. This handle was retrieved by a call to the **OleLockServer** function.

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_HANDLE
OLE_WAIT_FOR_RELEASE

Comments When the **OleLockServer** function is called more than once for a given server, the server's lock count is incremented. Each call to **OleUnlockServer** decrements the lock count. The server remains locked until the lock count is zero.

If the **OleUnlockServer** function returns OLE_WAIT_FOR_RELEASE, the application should call the **OleQueryReleaseStatus** function to determine whether the unlocking process has finished. In the call to **OleQueryReleaseStatus**, the application can cast the server handle to a long pointer to an object linking and embedding (OLE) object (**LPOLEOBJECT**):

```
OleQueryReleaseStatus((LPOLEOBJECT) lhserver);
```

When **OleQueryReleaseStatus** no longer returns OLE_BUSY, the server has been unlocked.

See Also **OleLockServer, OleQueryReleaseStatus**

OleUpdate 3.1

Syntax #include <ole.h>
OLESTATUS OleUpdate(lpObject)

function OleUpdate(Self: POleObject): TOleStatus;

The **OleUpdate** function updates the specified object. This function updates the presentation of the object and ensures that the object is up-to-date with respect to any linked objects it contains.

Parameters *lpObject* Points to the object to be updated.

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_BUSY
OLE_ERROR_OBJECT
OLE_ERROR_STATIC
OLE_WAIT_FOR_RELEASE

See Also **OleQueryOutOfDate**

OpenDriver 3.1

Syntax HDRVR OpenDriver(lpDriverName, lpSectionName, lParam)

function OpenDriver(DriverName, SectionName: PChar; lParam2: Longint): THandle;

The **OpenDriver** function performs necessary initialization operations such as setting members in installable-driver structures to their default values.

Parameters

lpDriverName	Points to a null-terminated string that specifies the name of an installable driver.
lpSectionName	Points to a null-terminated string that specifies the name of a section in the SYSTEM.INI file.
lParam	Specifies driver-specific information.

Return Value The return value is a handle of the installable driver, if the function is successful. Otherwise it is NULL.

Comments The string to which *lpDriverName* points must be identical to the name of the installable driver as it appears in the SYSTEM.INI file.

If the name of the installable driver appears in the [driver] section of the SYSTEM.INI file, the string pointed to by *lpSectionName* should be NULL. Otherwise this string should specify the name of the section in SYSTEM.INI that contains the driver name.

When an application opens a driver for the first time, Windows calls the **DriverProc** function with the DRV_LOAD, DRV_ENABLE, and DRV_OPEN messages. When subsequent instances of the driver are opened, only DRV_OPEN is sent.

The value specified in the *lParam* parameter is passed to the *lParam2* parameter of the **DriverProc** function.

See Also **CloseDriver, DriverProc**

PrintDlg 3.1

Syntax #include <commdlg.h>
BOOL PrintDlg(lppd)

function PrintDlg(var PrintDlg: TPrintDlg): Bool;

The **PrintDlg** function displays a Print dialog box or a Print Setup dialog box. The Print dialog box makes it possible for the user to specify the properties of a particular print job. The Print Setup dialog box makes it possible for the user to select additional job properties and configure the printer.

Parameters *lppd* Points to a **PRINTDLG** structure that contains information used to initialize the dialog box. When the **PrintDlg** function returns, this structure contains information about the user's selections.

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

```
#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;
```

Return Value The return value is nonzero if the function successfully configures the printer. The return value is zero if an error occurs, if the user chooses the Cancel button, or if the user chooses the Close command on the System menu to close the dialog box. (The return value is also zero if the user chooses the Setup button to display the Print Setup dialog box, chooses the OK button in the Print Setup dialog box, and then chooses the Cancel button in the Print dialog box.)

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

CDERR_FINDRESFAILURE	PDERR_CREATEICFAILURE
CDERR_INITIALIZATION	PDERR_DEFAULTDIFFERENT
CDERR_LOADRESFAILURE	PDERR_DNDMMISMATCH
CDERR_LOADSTRFAILURE	PDERR_GETDEVMODEFAIL
CDERR_LOCKRESFAILURE	PDERR_INITFAILURE
CDERR_MEMALLOCFAILURE	PDERR_LOADDRVFAILURE
CDERR_MEMLOCKFAILURE	PDERR_NODEFAULTPRN
CDERR_NOHINSTANCE	PDERR_NODEVICES
CDERR_NOHOOK	PDERR_PARSEFAILURE
CDERR_NOTEMPLATE	PDERR_PRINTERNOTFOUND
CDERR_STRUCTSIZE	PDERR_RETDEFFAILURE
	PDERR_SETUPFAILURE

Example The following example initializes the **PRINTDLG** structure, calls the **PrintDlg** function to display the Print dialog box, and prints a sample page of text if the return value is nonzero:

```
PRINTDLG pd;

/* Set all structure fields to zero. */

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

/* Initialize the necessary PRINTDLG structure fields. */

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

/* Print a test page if successful */

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

    /* Print text and rectangle */

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

QueryAbort 3.1

Syntax BOOL QueryAbort(hdc, reserved)

function QueryAbort(DC: HDC; Reserved: Integer): Bool;

The **QueryAbort** function calls the **AbortProc** callback function for a printing application and queries whether the printing should be terminated.

Parameters

hdc	Identifies the device context.
message	Specifies a reserved value. It should be zero.

Return Value The return value is TRUE if printing should continue or if there is no abort procedure. It is FALSE if the print job should be terminated. The return value is supplied by the **AbortProc** callback function.

See Also **AbortDoc, AbortProc, SetAbortProc**

QuerySendMessage 3.1

Syntax BOOL QuerySendMessage(hreserved1, hreserved2, hreserved3, lpMessage)

function QuerySendMessage(h1, h2, h3: THandle; lpmsg: PMsg): Bool;

The **QuerySendMessage** function determines whether a message sent by **SendMessage** originated from within the current task. If the message is an intertask message, **QuerySendMessage** puts it into the specified **MSG** structure.

Parameters

hreserved1	Reserved; must be NULL.
hreserved2	Reserved; must be NULL.
hreserved3	Reserved; must be NULL.
lpMessage	Specifies the **MSG** structure in which to place an intertask message. The **MSG** structure has the following form:

```
typedef struct tagMSG {     /* msg */
    HWND   hwnd;
    UINT   message;
    WPARAM wParam;
    LPARAM lParam;
    DWORD  time;
    POINT  pt;
} MSG;
```

Return Value The return value is zero if the message originated within the current task. Otherwise, it is nonzero.

Comments If the Windows debugger is entering soft mode, the application being debugged should reply to intertask messages by using the **ReplyMessage** function.

The NULL parameters are reserved for future use.

See Also **SendMessage, ReplyMessage**

RedrawWindow 3.1

Syntax BOOL RedrawWindow(hwnd, lprcUpdate, hrgnUpdate, fuRedraw)

function RedrawWindow(Wnd: HWnd; UpdateRect: PRect; UpdateRgn: HRgn; Flags: Word): Bool;

The **RedrawWindow** function updates the specified rectangle or region in the given window's client area.

Parameters

hwnd Identifies the window to be redrawn. If this parameter is NULL, the desktop window is updated.

lprcUpdate Points to a **RECT** structure containing the coordinates of the update rectangle. This parameter is ignored if the *hrgnUpdate* parameter contains a valid region handle. The **RECT** structure has the following form:

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

hrgnUpdate Identifies the update region. If both the *hrgnUpdate* and *lprcUpdate* parameters are NULL, the entire client area is added to the update region.

fuRedraw Specifies one or more redraw flags. This parameter can be a combination of flags:

The following flags are used to invalidate the window:

Value	Meaning
RDW_ERASE	Causes the window to receive a WM_ERASEBKGND message when the window is repainted. The RDW_INVALIDATE flag must also be specified; otherwise, RDW_ERASE has no effect.
RDW_FRAME	Causes any part of the non-client area of the window that intersects the update region to receive a WM_NCPAINT message. The RDW_INVALIDATE flag must also be specified; otherwise, RDW_FRAME has no effect. The WM_NCPAINT message is typically not sent during the execution of the **RedrawWindow** function unless either RDW_UPDATENOW or RDW_ERASENOW is specified.
RDW_INTERNALPAINT	Causes a WM_PAINT message to be posted to the window regardless of whether the window contains an invalid region.
RDW_INVALIDATE	Invalidate *lprcUpdate* or *hrgnUpdate* (only one may be non-NULL). If both are NULL, the entire window is invalidated.

The following flags are used to validate the window:

Value	Meaning
RDW_NOERASE	Suppresses any pending WM_ERASEBKGND messages.
RDW_NOFRAME	Suppresses any pending WM_NCPAINT messages. This flag must be used with RDW_VALIDATE and is typically used with RDW_NOCHILDREN. This option should be used with care, as it could cause parts of a window from painting properly.

Value	Meaning
RDW_NOINTERNALPAINT	Suppresses any pending internal WM_PAINT messages. This flag does not affect WM_PAINT messages resulting from invalid areas.
RDW_VALIDATE	Validates *lprcUpdate* or *hrgnUpdate* (only one may be non-NULL). If both are NULL, the entire window is validated. This flag does not affect internal WM_PAINT messages.

The following flags control when repainting occurs. No painting is performed by the **RedrawWindow** function unless one of these bits is specified.

Value	Meaning
RDW_ERASENOW	Causes the affected windows (as specified by the RDW_ALLCHILDREN and RDW_NOCHILDREN flags) to receive WM_NCPAINT and WM_ERASEBKGND messages, if necessary, before the function returns. WM_PAINT messages are deferred.
RDW_UPDATENOW	Causes the affected windows (as specified by the RDW_ALLCHILDREN and RDW_NOCHILDREN flags) to receive WM_NCPAINT, WM_ERASEBKGND, and WM_PAINT messages, if necessary, before the function returns.

By default, the windows affected by the **RedrawWindow** function depend on whether the specified window has the WS_CLIPCHILDREN style. The child windows of WS_CLIPCHILDREN windows are not affected; however, non-WS_CLIPCHILDREN windows are recursively validated or invalidated until a WS_CLIPCHILDREN window is encountered. The following flags control which windows are affected by the **RedrawWindow** function:

Value	Meaning
RDW_ALLCHILDREN	Includes child windows, if any, in the repainting operation.
RDW_NOCHILDREN	Excludes child windows, if any, from the repainting operation.

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

Comments When the **RedrawWindow** function is used to invalidate part of the desktop window, the desktop window does not receive a WM_PAINT message. To repaint the desktop, an application should use the RDW_ERASE flag to generate a WM_ERASEBKGND message.

See Also **GetUpdateRect, GetUpdateRgn, InvalidateRect, InvalidateRgn, UpdateWindow**

RegCloseKey 3.1

Syntax #include <shellapi.h>
LONG RegCloseKey(hkey)

function RegCloseKey(Key: HKey): Longint;

The **RegCloseKey** function closes a key. Closing a key releases the key's handle. When all keys are closed, the registration database is updated.

Parameters *hkey* Identifies the open key to close.

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

Comments The **RegCloseKey** function should be called only if a key has been opened by either the **RegOpenKey** function or the **RegCreateKey** function. The handle for a given key should not be used after it has been closed, because it may no longer be valid. Key handles should not be left open any longer than necessary.

Example The following example uses the **RegCreateKey** function to create the handle of a protocol, uses the **RegSetValue** function to set up the subkeys of the protocol, and then calls **RegCloseKey** to save the information in the database:

```
HKEY hkProtocol;

if (RegCreateKey(HKEY_CLASSES_ROOT,              /* root            */
    "NewAppDocument\\protocol\\StdFileEditing", /* protocol string */
    &hkProtocol) != ERROR_SUCCESS)          /* protocol key handle */
        return FALSE;

RegSetValue(hkProtocol,         /* handle of protocol key        */
    "server",                   /* name of subkey                */
    REG_SZ,                     /* required                      */
    "newapp.exe",               /* command to activate server    */
    10);                        /* text string size              */

RegSetValue(hkProtocol,         /* handle of protocol key        */
    "verb\\0",                  /* name of subkey                */
    REG_SZ,                     /* required                      */
    "EDIT",                     /* server should edit object     */
    4);                         /* text string size              */

RegCloseKey(hkProtocol);    /* closes protocol key and subkeys    */
```

See Also **RegCreateKey, RegDeleteKey, RegOpenKey, RegSetValue**

RegCreateKey 3.1

Syntax #include <shellapi.h>
LONG RegCreateKey(hkey, lpszSubKey, lphkResult)

function RegCreateKey(Key: HKey; SubKey: PChar; var Result: HKey): Longint;

The **RegCreateKey** function creates the specified key. If the key already exists in the registration database, **RegCreateKey** opens it.

Parameters

hkey	Identifies an open key (which can be HKEY_CLASSES_ROOT). The key opened or created by the **RegCreateKey** function is a subkey of the key identified by the *hkey* parameter. This value should not be NULL.
lpszSubKey	Points to a null-terminated string specifying the subkey to open or create.
lphkResult	Points to the handle of the key that is opened or created.

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

Comments An application can create keys that are subordinate to the top level of the database by specifying HKEY_CLASSES_ROOT for the *hKey* parameter. An application can use the **RegCreateKey** function to create several keys at once. For example, an application could create a subkey four levels deep and the three preceding subkeys by specifying a string of the following form for the *lpszSubKey* parameter:

subkey1\subkey2\subkey3\subkey4

Example The following example uses the **RegCreateKey** function to create the handle of a protocol, uses the **RegSetValue** function to set up the subkeys of the protocol, and then calls **RegCloseKey** to save the information in the database:

```
HKEY hkProtocol;

if (RegCreateKey(HKEY_CLASSES_ROOT,             /* root            */
    "NewAppDocument\\protocol\\StdFileEditing", /* protocol string */
    &hkProtocol) != ERROR_SUCCESS)          /* protocol key handle */
        return FALSE;

RegSetValue(hkProtocol,             /* handle of protocol key       */
    "server",                       /* name of subkey               */
    REG_SZ,                         /* required                     */
    "newapp.exe",                   /* command to activate server   */
    10);                            /* text string size             */

RegSetValue(hkProtocol,             /* handle of protocol key       */
    "verb\\0",                      /* name of subkey               */
    REG_SZ,                         /* required                     */
    "EDIT",                         /* server should edit object    */
    4);                             /* text string size             */

RegCloseKey(hkProtocol);    /* closes protocol key and subkeys      */
```

See Also **RegCloseKey, RegOpenKey, RegSetValue**

RegDeleteKey 3.1

Syntax #include <shellapi.h>
LONG RegDeleteKey(hkey, lpszSubKey)

function RegDeleteKey(Key: HKey; SubKey: PChar): Longint;

The **RegDeleteKey** function deletes the specified key. When a key is deleted, its value and all of its subkeys are deleted.

Parameters

hkey Identifies an open key (which can be HKEY_CLASSES_ROOT). The key deleted by the **RegDeleteKey** function is a subkey of this key.

lpszSubKey Points to a null-terminated string specifying the subkey to delete. This value should not be NULL.

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

If the error value is ERROR_ACCESS_DENIED, either the application does not have delete privileges for the specified key or another application has opened the specified key.

Example The following example uses the **RegQueryValue** function to retrieve the name of an object handler and then calls the **RegDeleteKey** function to delete the key if its value is nwappobj.dll:

```
char szBuff[80];
LONG cb;
HKEY hkStdFileEditing;

if (RegOpenKey(HKEY_CLASSES_ROOT,
        "NewAppDocument\\protocol\\StdFileEditing",
        &hkStdFileEditing) == ERROR_SUCCESS) {

    cb = sizeof(szBuff);


if (RegQueryValue(hkStdFileEditing,
            "handler",
            szBuff,
            &cb) == ERROR_SUCCESS
            && lstrcmpi("nwappobj.dll", szBuff) == 0)
        RegDeleteKey(hkStdFileEditing, "handler");
    RegCloseKey(hkStdFileEditing);
}
```

See Also **RegCloseKey**

RegEnumKey 3.1

Syntax #include <shellapi.h>
LONG RegEnumKey(hkey, iSubkey, lpszBuffer, cbBuffer)

function RegEnumKey(Key: HKey; index: Longint; buffer: PChar; cb: Longint): Longint;

The **RegEnumKey** function enumerates the subkeys of a specified key.

Parameters

hkey	Identifies an open key (which can be HKEY_CLASSES_ROOT) for which subkey information is retrieved.
iSubkey	Specifies the index of the subkey to retrieve. This value should be zero for the first call to the **RegEnumKey** function.
lpszBuffer	Points to a buffer that contains the name of the subkey when the function returns. This function copies only the name of the subkey, not the full key hierarchy, to the buffer.
cbBuffer	Specifies the size, in bytes, of the buffer pointed to by the *lpszBuffer* parameter.

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

Comments The first parameter of the **RegEnumKey** function must specify an open key. Applications typically precede the call to the **RegEnumKey** function with a call to the **RegOpenKey** function and follow it with a call to the **RegCloseKey** function. Calling **RegOpenKey** and **RegCloseKey** is not necessary when the first parameter is HKEY_CLASSES_ROOT, because this key is always open and available; however, calling **RegOpenKey** and **RegCloseKey** in this case is a time optimization. While an application is using the **RegEnumKey** function, it should not make calls to any registration functions that might change the key being queried.

To enumerate subkeys, an application should initially set the *iSubkey* parameter to zero and then increment it on successive calls.

Example The following example uses the **RegEnumKey** function to put the values associated with top-level keys into a list box:

```
HKEY hkRoot;
char szBuff[80], szValue[80];
static DWORD dwIndex;
LONG cb;

if (RegOpenKey(HKEY_CLASSES_ROOT, NULL, &hkRoot) == ERROR_SUCCESS) {
    for (dwIndex = 0; RegEnumKey(hkRoot, dwIndex, szBuff,
             sizeof(szBuff)) == ERROR_SUCCESS; ++dwIndex) {
        if (*szBuff == '.')
            continue;
        cb = sizeof(szValue);
        if (RegQueryValue(hkRoot, (LPSTR) szBuff, szValue,
                &cb) == ERROR_SUCCESS)
            SendDlgItemMessage(hDlg, ID_ENUMLIST, LB_ADDSTRING, 0,
                (LONG) (LPSTR) szValue);
    }
    RegCloseKey(hkRoot);
}
```

See Also **RegQueryValue**

RegOpenKey 3.1

Syntax #include <shellapi.h>
LONG RegOpenKey(hkey, lpszSubKey, lphkResult)

function RegOpenKey(Key: HKey; SubKey: PChar; var Result: HKey): Longint;

The **RegOpenKey** function opens the specified key.

Parameters

hkey	Identifies an open key (which can be HKEY_CLASSES_ROOT). The key opened by the **RegOpenKey** function is a subkey of the key identified by this parameter. This value should not be NULL.
lpszSubKey	Points to a null-terminated string specifying the name of the subkey to open.
lphkResult	Points to the handle of the key that is opened.

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

Comments Unlike the **RegCreateKey** function, the **RegOpenKey** function does not create the specified key if the key does not exist in the database.

Example The following example uses the **RegOpenKey** function to retrieve the handle of the StdFileEditing subkey, calls the **RegQueryValue** function to retrieve the name of an object handler, and then calls the **RegDeleteKey** function to delete the key if its value is nwappobj.dll:

```
char szBuff[80];
LONG cb;
HKEY hkStdFileEditing;

if (RegOpenKey(HKEY_CLASSES_ROOT,
        "NewAppDocument\\protocol\\StdFileEditing",
        &hkStdFileEditing) == ERROR_SUCCESS) {

    cb = sizeof(szBuff);
    if (RegQueryValue(hkStdFileEditing,
            "handler",
            szBuff,
            &cb) == ERROR_SUCCESS
            && lstrcmpi("nwappobj.dll", szBuff) == 0)
        RegDeleteKey(hkStdFileEditing, "handler");
    RegCloseKey(hkStdFileEditing);
}
```

See Also **RegCreateKey**

RegQueryValue 3.1

Syntax #include <shellapi.h>
LONG RegQueryValue(hkey, lpszSubKey, lpszValue, lpcb)

function RegQueryValue(Key: HKey; SubKey: PChar; Value: PChar; var cb: Longint): Longint;

The **RegQueryValue** function retrieves the text string associated with a specified key.

Parameters

hkey	Identifies a currently open key (which can be HKEY_CLASSES_ROOT). This value should not be NULL.
lpszSubKey	Points to a null-terminated string specifying the name of the subkey of the *hkey* parameter for which a text string is retrieved. If this parameter is NULL or points to an empty string, the function retrieves the value of the *hkey* parameter.
lpszValue	Points to a buffer that contains the text string when the function returns.

lpcb — Points to a variable specifying the size, in bytes, of the buffer pointed to by the *lpszValue* parameter. When the function returns, this variable contains the size of the string copied to *lpszValue*, including the null-terminating character.

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

Example The following example uses the **RegOpenKey** function to retrieve the handle of the StdFileEditing subkey, calls the **RegQueryValue** function to retrieve the name of an object handler and then calls the **RegDeleteKey** function to delete the key if its value is nwappobj.dll:

```
char szBuff[80];
LONG cb;
HKEY hkStdFileEditing;

if (RegOpenKey(HKEY_CLASSES_ROOT,
        "NewAppDocument\\protocol\\StdFileEditing",
        &hkStdFileEditing) == ERROR_SUCCESS) {

    cb = sizeof(szBuff);

if (RegQueryValue(hkStdFileEditing,
            "handler",
            szBuff,
            &cb) == ERROR_SUCCESS
            && lstrcmpi("nwappobj.dll", szBuff) == 0)
        RegDeleteKey(hkStdFileEditing, "handler");
    RegCloseKey(hkStdFileEditing);
}
```

See Also **RegEnumKey**

RegSetValue 3.1

Syntax #include <shellapi.h>
LONG RegSetValue(hkey, lpszSubKey, fdwType, lpszValue, cb)

function RegSetValue(Key: HKey; SubKey: PChar; ValType: Longint; Value: PChar; cb: Longint): Longint;

The **RegSetValue** function associates a text string with a specified key.

Parameters

hkey — Identifies a currently open key (which can be HKEY_CLASSES_ROOT). This value should not be NULL.

lpszSubKey	Points to a null-terminated string specifying the subkey of the *hkey* parameter with which a text string is associated. If this parameter is NULL or points to an empty string, the function sets the value of the *hkey* parameter.
fdwType	Specifies the string type. For Windows version 3.1, this value must be REG_SZ.
lpszValue	Points to a null-terminated string specifying the text string to set for the given key.
cb	Specifies the size, in bytes, of the string pointed to by the *lpszValue* parameter. For Windows version 3.1, this value is ignored.

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

Comments If the key specified by the *lpszSubKey* parameter does not exist, the **RegSetValue** function creates it.

Example The following example uses the **RegSetValue** function to register a filename extension and its associated class name:

```
RegSetValue(HKEY_CLASSES_ROOT,  /* root                          */
    ".XXX",                     /* string for filename extension */
    REG_SZ,                     /* required                      */
    "NewAppDocument",           /* class name for extension      */
    14);                        /* size of text string           */

RegSetValue(HKEY_CLASSES_ROOT,  /* root                          */
    "NewAppDocument",           /* string for class-definition key */
    REG_SZ,                     /* required                      */
    "New Application",          /* text description of class     */
    15);                        /* size of text string           */
```

See Also **RegCreateKey, RegQueryValue**

ReplaceText 3.1

Syntax #include <commdlg.h>
HWND ReplaceText(lpfr)

function ReplaceText(var FindReplace: TFindReplace): HWnd;

The **ReplaceText** function creates a system-defined modeless dialog box that makes it possible for the user to find and replace text within a document. The application must perform the actual find and replace operations.

Parameters *lpfr*

Points to a **FINDREPLACE** structure that contains information used to initialize the dialog box. When the user makes a selection in the dialog box, the system fills this structure with information about the user's selection and then sends a message to the application. This message contains a pointer to the **FINDREPLACE** structure.

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

```
#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;
```

Return Value The return value is the window handle of the dialog box, or it is NULL if an error occurs. An application can use this handle to communicate with or to close the dialog box.

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

CDERR_FINDRESFAILURE
CDERR_INITIALIZATION
CDERR_LOADRESFAILURE
CDERR_LOADSTRFAILURE
CDERR_LOCKRESFAILURE
CDERR_MEMALLOCFAILURE
CDERR_MEMLOCKFAILURE
CDERR_NOHINSTANCE
CDERR_NOHOOK
CDERR_NOTEMPLATE
CDERR_STRUCTSIZE
FRERR_BUFFERLENGTHZERO

Comments The dialog box procedure for the **ReplaceText** function passes user requests to the application through special messages. The *lParam* parameter of each of these messages contains a pointer to a **FINDREPLACE** structure. The procedure sends the messages to the window identified by the **hwndOwner** member of the **FINDREPLACE** structure. An application can register the identifier for these messages by specifying the commdlg_FindReplace string in a call to the **RegisterWindowMessage** function.

For the TAB key to function correctly, any application that calls the **ReplaceText** function must also call the **IsDialogMessage** function in its main message loop. (The **IsDialogMessage** function returns a value that indicates whether messages are intended for the Replace dialog box.)

Example This example initializes a **FINDREPLACE** structure and calls the **ReplaceText** function to display the Replace dialog box:

```
FINDREPLACE fr;
char szFindWhat[256] = "";       /* string to find    */
char szReplaceWith[256] = "";    /* string to replace */

/* Set all structure fields to zero. */

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

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

hDlg = ReplaceText(&fr);
```

In addition to initializing the members of the **FINDREPLACE** structure and calling the **ReplaceText** function, an application must register the special FINDMSGSTRING message and process messages from the dialog box. Refer to the description of the **FindText** function for an example that shows how an application registers and processes a message.

See Also **FindText, IsDialogMessage, RegisterWindowMessage**

ResetDC 3.1

Syntax #include <print.h>
HDC ResetDC(hdc, lpdm)

function ResetDC(aHdc: HDC; DevMode: PDevMode): HDC;

The **ResetDC** function updates the given device context, based on the information in the specified **DEVMODE** structure.

Parameters

hdc Identifies the device context to be updated.

lpdm Points to a **DEVMODE** structure containing information about the new device context. The **DEVMODE** structure has the following form:

```
#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;
```

Return Value The return value is the handle of the original device context if the function is successful. Otherwise, it is NULL.

Comments An application will typically use the **ResetDC** function when a window receives a WM_DEVMODECHANGE message. **ResetDC** can also be used to change the paper orientation or paper bins while printing a document.

The **ResetDC** function cannot be used to change the driver name, device name or the output port. When the user changes the port connection or device name, the application must delete the original device context and create a new device context with the new information.

Before calling **ResetDC**, the application must ensure that all objects (other than stock objects) that had been selected into the device context have been selected out.

See Also **DeviceCapabilities, Escape, ExtDeviceMode**

ScaleViewportExtEx 3.1

Syntax BOOL ScaleViewportExtEx(hdc, nXnum, nXdenom, nYnum, nYdenom, lpSize)

function ScaleViewportExtEx(DC: HDC; Xnum, Xdenom, Ynum, Ydenom: Integer; Size: PSize): Bool;

The **ScaleViewportExtEx** function modifies the viewport extents relative to the current values. The formulas are written as follows:

```
xNewVE = (xOldVE * Xnum) / Xdenom
yNewVE = (yOldVE * Ynum) / Ydenom
```

The new extent is calculated by multiplying the current extents by the given numerator and then dividing by the given denominator.

Parameters

hdc	Identifies the device context.
nXnum	Specifies the amount by which to multiply the current x-extent.
nXdenom	Specifies the amount by which to divide the current x-extent.
nYnum	Specifies the amount by which to multiply the current y-extent.
nYdenom	Specifies the amount by which to divide the current y-extent.
lpSize	Points to a **SIZE** structure. The previous viewport extents, in device units, are placed in this structure. If *lpSize* is NULL, nothing is returned.

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

ScaleWindowExtEx 3.1

Syntax BOOL ScaleWindowExtEx(hdc, nXnum, nXdenom, nYnum, nYdenom, lpSize)

function ScaleWindowExtEx(DC: HDC; Xnum, Xdenom, Ynum, Ydenom: Integer; Size: PSize): Bool;

The **ScaleWindowExtEx** function modifies the window extents relative to the current values. The formulas are written as follows:

```
xNewWE = (xOldWE * Xnum) / Xdenom
yNewWE = (yOldWE * Ynum) / Ydenom
```

The new extent is calculated by multiplying the current extents by the given numerator and then dividing by the given denominator.

Parameters

hdc	Identifies the device context.
nXnum	Specifies the amount by which to multiply the current x-extent.
nXdenom	Specifies the amount by which to divide the current x-extent.
nYnum	Specifies the amount by which to multiply the current y-extent.
nYdenom	Specifies the amount by which to divide the current y-extent.
lpSize	Points to a **SIZE** structure. The previous window extents, in logical units, are placed in this structure. If *lpSize* is NULL, nothing is returned.

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

ScrollWindowEx 3.1

Syntax int ScrollWindowEx(hwnd, dx, dy, lprcScroll, lprcClip, hrgnUpdate, lprcUpdate, fuScroll)

function ScrollWindowEx(Wnd: HWnd; dx, dy: Integer; Scroll, Clip: PRect; UpdateRgn: HRgn; UpdateRect: PRect; Flags: Word): Integer;

The **ScrollWindowEx** function scrolls the contents of a window's client area. This function is similar to the **ScrollWindow** function, with some additional features.

Parameters

hwnd — Identifies the window to be scrolled.

dx — Specifies the amount, in device units, of horizontal scrolling. This parameter must be a negative value to scroll to the left.

dy — Specifies the amount, in device units, of vertical scrolling. This parameter must be a negative value to scroll up.

lprcScroll — Points to a **RECT** structure that specifies the portion of the client area to be scrolled. If this parameter is NULL, the entire client area is scrolled. The **RECT** structure has the following form:

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

lprcClip — Points to a **RECT** structure that specifies the clipping rectangle to scroll. This structure takes precedence over the rectangle pointed to by the *lprcScroll* parameter. Only bits inside this rectangle are scrolled. Bits outside this rectangle are not affected even if they are in the *lprcScroll* rectangle. If this parameter is NULL, the entire client area is scrolled.

hrgnUpdate — Identifies the region that is modified to hold the region invalidated by scrolling. This parameter may be NULL.

lprcUpdate — Points to a **RECT** structure that will receive the boundaries of the rectangle invalidated by scrolling. This parameter may be NULL.

fuScroll — Specifies flags that control scrolling. This parameter can be one of the following values:

Value	Meaning
SW_ERASE	When specified with SW_INVALIDATE, erases the newly invalidated region by sending a WM_ERASEBKGND message to the window.
SW_INVALIDATE	Invalidates the region identified by the *hrgnUpdate* parameter after scrolling.

Value	Meaning
SW_SCROLLCHILDREN	Scrolls all child windows that intersect the rectangle pointed to by *lprcScroll* by the number of pixels specified in the *dx* and *dy* parameters. Windows sends a WM_MOVE message to all child windows that intersect *lprcScroll*, even if they do not move. The caret is repositioned when a child window is scrolled and the cursor rectangle intersects the scroll rectangle.

Return Value The return value is SIMPLEREGION (rectangular invalidated region), COMPLEXREGION (nonrectangular invalidated region; overlapping rectangles), or NULLREGION (no invalidated region), if the function is successful. Otherwise, the return value is ERROR.

Comments If SW_INVALIDATE and SW_ERASE are not specified, **ScrollWindowEx** does not invalidate the area that is scrolled away from. If either of these flags is set, **ScrollWindowEx** invalidates this area. The area is not updated until the application calls the **UpdateWindow** function, calls the **RedrawWindow** function (specifying RDW_UPDATENOW or RDW_ERASENOW), or retrieves the WM_PAINT message from the application queue.

If the window has the WS_CLIPCHILDREN style, the returned areas specified by *hrgnUpdate* and *lprcUpdate* represent the total area of the scrolled window that must be updated, including any areas in child windows that need qupdating.

If the SW_SCROLLCHILDREN flag is specified, Windows will not properly update the screen if part of a child window is scrolled. The part of the scrolled child window that lies outside the source rectangle will not be erased and will not be redrawn properly in its new destination. Use the **DeferWindowPos** function to move child windows that do not lie completely within the *lprcScroll* rectangle.

All input and output coordinates (for *lprcScroll*, *lprcClip*, *lprcUpdate*, and *hrgnUpdate*) are assumed to be in client coordinates, regardless of whether the window has the CS_OWNDC or CS_CLASSDC class style. Use the **LPtoDP** and **DPtoLP** functions to convert to and from logical coordinates, if necessary.

See Also **RedrawWindow, ScrollDC, ScrollWindow, UpdateWindow**

SendDriverMessage 3.1

Syntax LRESULT SendDriverMessage(hdrvr, msg, lParam1, lParam2)

function SendDriverMessage(Driver: THandle; message: Word; lParam1, lParam2: Longint): Longint;

The **SendDriverMessage** function sends the specified message to the given installable driver.

Parameters

hdrvr Identifies the installable driver.

msg Specifies the message that the driver must process. The following messages should never be sent by an application directly to the driver; they are sent only by the system:

DRV_CLOSE
DRV_DISABLE
DRV_ENABLE
DRV_EXITAPPLICATION
DRV_EXITSESSION
DRV_FREE
DRV_LOAD
DRV_OPEN

lParam1 Specifies 32 bits of additional message-dependent information.

lParam2 Specifies 32 bits of additional message-dependent information.

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

See Also **DefDriverProc**

SetAbortProc 3.1

Syntax int SetAbortProc(hdc, abrtprc)

function SetAbortProc(DC: HDC; AbortProc: TAbortProc): Integer;

The **SetAbortProc** function sets the application-defined procedure that allows a print job to be canceled during spooling. This function replaces the SETABORTPROC printer escape for Windows version 3.1.

Parameters

hdc	Identifies the device context for the print job.
abrtprc	Specifies the procedure-instance address of the callback function. The address must have been created by using the **MakeProcInstance** function. For more information about the callback function, see the description of the **AbortProc** callback function.

Return Value The return value is greater than zero if the function is successful. Otherwise, it is less than zero.

See Also **AbortDoc, AbortProc, Escape**

SetBitmapDimensionEx 3.1

Syntax BOOL SetBitmapDimensionEx(hbm, nX, nY, lpSize)

function SetBitmapDimensionEx(BM: HBitmap; nX, nY: Integer; Size: PSize): Bool;

The **SetBitmapDimensionEx** function assigns the preferred size to a bitmap, in 0.1-millimeter units. The graphics device interface (GDI) does not use these values, except to return them when an application calls the **GetBitmapDimensionEx** function.

Parameters

hbm	Identifies the bitmap.
nX	Specifies the width of the bitmap, in 0.1-millimeter units.
nY	Specifies the height of the bitmap, in 0.1-millimeter units.
lpSize	Points to a **SIZE** structure. The previous bitmap dimensions are placed in this structure. If *lpSize* is NULL, nothing is returned. The **SIZE** structure has the following form:

```
typedef struct tagSIZE {
    int cx;
    int cy;
} SIZE;
```

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

SetBoundsRect 3.1

Syntax UINT SetBoundsRect(hdc, lprcBounds, flags)

function SetBoundsRect(DC: HDC; var Bounds: TRect; Flags: Word): Word;

The **SetBoundsRect** function controls the accumulation of bounding-rectangle information for the specified device context.

Parameters

hdc Identifies the device context to accumulate bounding rectangles for.

lprcBounds Points to a **RECT** structure that is used to set the bounding rectangle. Rectangle dimensions are given in logical coordinates. This parameter can be NULL. The **RECT** structure has the following form:

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

flags Specifies how the new rectangle will be combined with the accumulated rectangle. This parameter may be a combination of the following values:

Value	Meaning
DCB_ACCUMULATE	Add the rectangle specified by the *lprcBounds* parameter to the bounding rectangle (using a rectangle union operation).
DCB_DISABLE	Turn off bounds accumulation.
DCB_ENABLE	Turn on bounds accumulation. (The default setting for bounds accumulation is disabled.)
DCB_RESET	Set the bounding rectangle empty.
DCB_SET	Set the bounding rectangle to the coordinates specified by the *lprcBounds* parameter.

Return Value The return value is the current state of the bounding rectangle, if the function is successful. Like the *flags* parameter, the return value can be a combination of DCB_ values.

Comments Windows can maintain a bounding rectangle for all drawing operations. This rectangle can be queried and reset by the application. The drawing bounds are useful for invalidating bitmap caches.

To ensure that a rectangle is empty, an application should check the DCB_ACCUMULATE and DCB_RESET flags in the return value. If the DCB_RESET flag is set and the DCB_ACCUMULATE flag is not set, the bounding rectangle is empty.

See Also **GetBoundsRect**

SetMetaFileBitsBetter 3.1

Syntax HGLOBAL SetMetaFileBitsBetter(hmf)

function SetMetaFileBitsBetter(mf: THandle): THandle;

The **SetMetaFileBitsBetter** function creates a memory metafile from the data in the specified global-memory object.

Parameters

hmf	Identifies the global-memory object that contains the metafile data. The object must have been created by a previous call to the **GetMetaFileBits** function.

Return Value The return value is the handle of a memory metafile, if the function is successful. Otherwise, the return value is NULL.

Comments The global-memory handle returned by **SetMetaFileBitsBetter** is owned by GDI, not by the application. This enables applications that use metafiles to support object linking and embedding (OLE) to use metafiles that persist beyond the termination of the application. An OLE application should always use **SetMetaFileBitsBetter** instead of the **SetMetaFileBits** function.

After the **SetMetaFileBitsBetter** function returns, the metafile handle returned by the function should be used to refer to the metafile, instead of the handle identified by the *hmf* parameter.

See Also **GetMetaFileBits, SetMetaFileBits**

SetSelectorBase 3.1

Syntax UINT SetSelectorBase(selector, dwBase)

function SetSelectorBase(Selector: Word; Base: Longint): Word;

The **SetSelectorBase** function sets the base and limit of a selector.

Parameters

selector	Specifies the new selector value.
dwBase	Specifies the new base value.

Return Value The return value is the new selector value, if the function is successful.

See Also **GetSelectorBase, GetSelectorLimit, SetSelectorLimit**

SetSelectorLimit 3.1

Syntax UINT SetSelectorLimit(selector, dwBase)

function SetSelectorLimit(Selector: Word; Base: Longint): Word;

The **SetSelectorLimit** function sets the limit of a selector.

Parameters

selector	Specifies the new selector value.
dwBase	Specifies the current base value for *selector*.

Return Value The return value is always zero.

See Also **GetSelectorBase, GetSelectorLimit, SetSelectorBase**

SetViewportExtEx 3.1

Syntax BOOL SetViewportExtEx(hdc, nX, nY, lpSize)

function SetViewportExtEx(DC: HDC; nX, nY: Integer; Size: PSize): Bool;

The **SetViewportExtEx** function sets the x- and y-extents of the viewport of the specified device context. The viewport, along with the window, defines how points are mapped from logical coordinates to device coordinates.

Parameters

hdc	Identifies the device context.

nX	Specifies the x-extent of the viewport, in device units.
nY	Specifies the y-extent of the viewport, in device units.
lpSize	Points to a **SIZE** structure. The previous extents of the viewport, in device units, are placed in this structure. If *lpSize* is NULL, nothing is returned. The **SIZE** structure has the following form:

```
typedef struct tagSIZE {
    int cx;
    int cy;
} SIZE;
```

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

Comments When the following mapping modes are set, calls to the **SetWindowExtEx** and **SetViewportExtEx** functions are ignored:

MM_HIENGLISH
MM_HIMETRIC
MM_LOENGLISH
MM_LOMETRIC
MM_TEXT
MM_TWIPS

When MM_ISOTROPIC mode is set, an application must call the **SetWindowExtEx** function before it calls **SetViewportExtEx**.

See Also **SetWindowExtEx**

SetViewportOrgEx 3.1

Syntax BOOL SetViewportOrgEx(hdc, nX, nY, lpPoint)

function SetViewportOrgEx(DC: HDC; nX, nY: Integer; Point: PPoint): Bool;

The **SetViewportOrgEx** function sets the viewport origin of the specified device context. The viewport, along with the window, defines how points are mapped from logical coordinates to device coordinates.

Parameters

hdc	Identifies the device context.
nX	Specifies the x-coordinate, in device units, of the origin of the viewport.

nY Specifies the y-coordinate, in device units, of the origin of the viewport.

lpPoint Points to a **POINT** structure. The previous origin of the viewport, in device coordinates, is placed in this structure. If *lpPoint* is NULL, nothing is returned. The **POINT** structure has the following form:

```
typedef struct tagPOINT {   /* pt */
   int x;
   int y;
} POINT;
```

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

See Also **SetWindowOrgEx**

SetWinDebugInfo 3.1

Syntax BOOL SetWinDebugInfo(lpwd)

function SetWinDebugInfo(DebugInfo: PWinDebugInfo): Bool;

The **SetWinDebugInfo** function sets current system-debugging information for the debugging version of the Windows 3.1 operating system.

Parameters *lpwdi* Points to a **WINDEBUGINFO** structure that specifies the type of debugging information to be set. The **WINDEBUGINFO** structure has the following form:

```
typedef struct tagWINDEBUGINFO {
    UINT    flags;
    DWORD   dwOptions;
    DWORD   dwFilter;
    char    achAllocModule[8];
    DWORD   dwAllocBreak;
    DWORD   dwAllocCount;
} WINDEBUGINFO;
```

Return Value The return value is nonzero if the function is successful. It is zero if the pointer specified in the *lpwdi* parameter is invalid, the **flags** member of the **WINDEBUGINFO** structure is invalid, or the function is not called in the debugging version of Windows 3.1.

Comments The **flags** member of the **WINDEBUGINFO** structure specifies which debugging information should be set. Applications need initialize only those members of the **WINDEBUGINFO** structure that correspond to the flags set in the **flags** member.

Changes to debugging information made by calling **SetWinDebugInfo** apply only until you exit the system or restart your computer.

See Also **GetWinDebugInfo**

SetWindowExtEx 3.1

Syntax BOOL SetWindowExtEx(hdc, nX, nY, lpSize)

function SetWindowExtEx(DC: HDC; nX, nY: Integer; Size: PSize): Bool;

The **SetWindowExtEx** function sets the x- and y-extents of the window associated with the specified device context. The window, along with the viewport, defines how points are mapped from logical coordinates to device coordinates.

Parameters

hdc	Identifies the device context.
nX	Specifies the x-extent, in logical units, of the window.
nY	Specifies the y-extent, in logical units, of the window.
lpSize	Points to a **SIZE** structure. The previous extents of the window (in logical units) are placed in this structure. If *lpSize* is NULL nothing is returned. The **SIZE** structure has the following form:

```
typedef struct tagSIZE {
    int cx;
    int cy;
} SIZE;
```

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

Comments When the following mapping modes are set, calls to the **SetWindowExtEx** and **SetViewportExt** functions are ignored:

MM_HIENGLISH
MM_HIMETRIC
MM_LOENGLISH

MM_LOMETRIC
MM_TEXT
MM_TWIPS

When MM_ISOTROPIC mode is set, an application must call the **SetWindowExtEx** function before calling **SetViewportExt**.

See Also **SetViewportExtEx**

SetWindowOrgEx 3.1

Syntax BOOL SetWindowOrgEx(hdc, nX, nY, lpPoint)

function SetWindowOrgEx(DC: HDC; nX, nY: Integer; Point: PPoint): Bool;

The **SetWindowOrgEx** function sets the window origin of the specified device context. The window, along with the viewport, defines how points are mapped from logical coordinates to device coordinates.

Parameters

hdc Identifies the device context.

nX Specifies the logical x-coordinate of the new origin of the window.

nY Specifies the logical y-coordinate of the new origin of the window.

lpPoint Points to a **POINT** structure. The previous origin of the window is placed in this structure. If *lpPoint* is NULL nothing is returned. The **POINT** structure has the following form:

```
typedef struct tagPOINT {   /* pt */
   int x;
   int y;
} POINT;
```

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

See Also **SetViewportOrgEx**

SetWindowPlacement 3.1

Syntax BOOL SetWindowPlacement(hwnd, lpwndpl)

function SetWindowPlacement(Wnd: HWnd; Placement: PWindowPlacement): Bool;

The **SetWindowPlacement** function sets the show state and the normal (restored), minimized, and maximized positions for a window.

Parameters

hwnd Identifies the window.

lpwndpl Points to a **WINDOWPLACEMENT** structure that specifies the new show state and positions. The **WINDOWPLACEMENT** structure has the following form:

```
typedef struct tagWINDOWPLACEMENT {     /* wndpl */
    UINT  length;
    UINT  flags;
    UINT  showCmd;
    POINT ptMinPosition;
    POINT ptMaxPosition;
    RECT  rcNormalPosition;
} WINDOWPLACEMENT;
```

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

See Also **GetWindowPlacement**

SetWindowsHookEx 3.1

Syntax HHOOK SetWindowsHookEx(idHook, hkprc, hinst, htask)

function SetWindowsHookEx(HookId: Integer; Hook: THookProc; Module, Task: THandle): HHook;

The **SetWindowsHookEx** function installs an application-defined hook function into a hook chain. This function is an extended version of the **SetWindowsHook** function.

Parameters *idHook* Specifies the type of hook to be installed. This parameter can be one of the following values:

Value	Meaning
WH_CALLWNDPROC	Installs a window-procedure filter. For more information, see the description of the **CallWndProc** callback function.
WH_CBT	Installs a computer-based training (CBT) filter. For more information, see the description of the **CBTProc** callback function.
WH_DEBUG	Installs a debugging filter. For more information, see the description of the **DebugProc** callback function.

Value	Meaning
WH_GETMESSAGE	Installs a message filter. For more information, see the description of the **GetMsgProc** callback function.
WH_HARDWARE	Installs a nonstandard hardware-message filter. For more information, see the description of the **HardwareProc** callback function.
WH_JOURNALPLAYBACK	Installs a journaling playback filter. For more information, see the description of the **JournalPlaybackProc** callback function.
WH_JOURNALRECORD	Installs a journaling record filter. For more information, see the description of the **JournalRecordProc** callback function.
WH_KEYBOARD	Installs a keyboard filter. For more information, see the description of the **KeyboardProc** callback function.
WH_MOUSE	Installs a mouse-message filter. For more information, see the description of the **MouseProc** callback function.
WH_MSGFILTER	Installs a message filter. For more information, see the description of the **MessageProc** callback function.
WH_SYSMSGFILTER	Installs a system-wide message filter. For more information, see the description of the **SysMsgProc** callback function.

hkprc Specifies the procedure-instance address of the application-defined hook procedure to be installed.

hinst Identifies the instance of the module containing the hook function.

htask Identifies the task for which the hook is to be installed. If this parameter is NULL, the installed hook function has system scope and may be called in the context of any process or task in the system.

Return Value The return value is a handle of the installed hook, if the function is successful. The application or library must use this handle to identify the hook when it calls the **CallNextHookEx** and **UnhookWindowsHookEx** functions. The return value is NULL if an error occurs.

Comments An application or library can use the **GetCurrentTask** or **GetWindowTask** function to obtain task handles for use in hooking a particular task.

Hook procedures used with **SetWindowsHookEx** must be declared as follows:

```
DWORD HookProc(code, wParam, lParam)
int code;
WORD wParam;
LONG lParam;
{
    if (...)
        return CallNextHookEx(hhook, code, wParam, lParam);
}

THookProc=function(Code: Integer; wParam: Word; lParam: Longint): Longint;
```

Chaining to the next hook procedure (that is, calling the **CallNextHookProc** function) is optional. An applicaiton or library can call the next hook procedure either before or after any processing in its own hook procedure.

Before terminating, an application must call the **UnhookWindowsHookEx** function to free system resources associated with the hook.

Some hooks may be set with system scope only, and others may be set only for a specific task, as shown in the following list:

Hook	Scope
WH_CALLWNDPROC	Task or system
WH_CBT	Task or system
WH_DEBUG	Task or system
WH_GETMESSAGE	Task or system
WH_HARDWARE	Task or system
WH_JOURNALRECORD	System only
WH_JOURNALPLAYBACK	System only
WH_KEYBOARD	Task or system
WH_MOUSE	Task or system
WH_MSGFILTER	Task or system
WH_SYSMSGFILTER	System only

For a given hook type, task hooks are called first, then system hooks.

The WH_CALLWNDPROC hook affects system performance. It is supplied for debugging purposes only.

The system hooks are a shared resource. Installing one affects all applications. All system hook functions must be in libraries. System hooks should be restricted to special-purpose applications or to use as a development aid during debugging of an application. Libraries that no longer need the hook should remove the filter function.

It is a good idea for several reasons to use task hooks rather than system hooks: They do not incur a system-wide overhead in applications that are not affected by the call (or that ignore the call); they do not require packaging the hook-procedure implementation in a separate dynamic-link library; they will continue to work even when future versions of Windows prevent applications from installing system-wide hooks for security reasons.

To install a filter function, the **SetWindowsHookEx** function must receive a procedure-instance address of the function and the function must be exported in the library's module-definition file. Libraries can pass the procedure address directly. Tasks must use the **MakeProcInstance** function to get a procedure-instance address. Dynamic-link libraries must use the **GetProcAddress** function to get a procedure-instance address.

For a given hook type, task hooks are called first, then system hooks.

The WH_SYSMSGFILTER hooks are called before the WH_MSGFILTER hooks. If any of the WH_SYSMSGFILTER hook functions return TRUE, the WH_MSGFILTER hooks are not called.

See Also **CallNextHookEx, GetProcAddress, MakeProcInstance, MessageBox, PeekMessage, PostMessage, SendMessage, UnhookWindowsHookEx**

ShellExecute 3.1

Syntax #include <shellapi.h>
HINSTANCE ShellExecute(hwnd, lpszOp, lpszFile, lpszParams, lpszDir, fsShowCmd)

function ShellExecute(hWnd: HWnd; Operation, FileName, Parameters, Directory: PChar; ShowCmd: Integer): THandle;

The **ShellExecute** function opens or prints the specified file.

Parameters

hwnd Identifies the parent window. This window receives any message boxes an application produces (for example, for error reporting).

lpszOp Points to a null-terminated string specifying the operation to perform. This string can be "open" or "print". If this parameter is NULL, "open" is the default value.

lpszFile Points to a null-terminated string specifying the file to open.

lpszParams Points to a null-terminated string specifying parameters passed to the application when the *lpszFile* parameter specifies an executable file. If *lpszFile* points to a string specifying a document file, this parameter is NULL.

lpszDir Points to a null-terminated string specifying the default directory.

fsShowCmd Specifies whether the application window is to be shown when the application is opened. This parameter can 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.

Value	Meaning
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).

Return Value The return value is the instance handle of the application that was opened or printed, if the function is successful. (This handle could also be the handle of a DDE server application.) A return value less than or equal to 32 specifies an error. The possible error values are listed in the following Comments section.

Errors The **ShellExecute** function returns the value 31 if there is no association for the specified file type or if there is no association for the specified action within the file type. The other possible error values are as follows:

Value	Meaning
0	System was out of memory, executable file was corrupt, or relocations were invalid.
2	File was not found.
3	Path was not found.
5	Attempt was made to dynamically link to a task, or there was a sharing or network-protection error.
6	Library required separate data segments for each task.
8	There was insufficient memory to start the application.
10	Windows version was incorrect.
11	Executable file was invalid. Either it was not a Windows application or there was an error in the .EXE image.
12	Application was designed for a different operating system.
13	Application was designed for MS-DOS 4.0.
14	Type of executable file was unknown.
15	Attempt was made to load a real-mode application (developed for an earlier version of Windows).

Value	Meaning
16	Attempt was made to load a second instance of an executable file containing multiple data segments that were not marked read-only.
19	Attempt was made to load a compressed executable file. The file must be decompressed before it can be loaded.
20	Dynamic-link library (DLL) file was invalid. One of the DLLs required to run this application was corrupt.
21	Application requires Microsoft Windows 32-bit extensions.

Comments The file specified by the *lpszFile* parameter can be a document file or an executable file. If it is a document file, this function opens or prints it, depending on the value of the *lpszOp* parameter. If it is an executable file, this function opens it, even if the string "print" is pointed to by *lpszOp*.

See Also **FindExecutable**

ShellProc 3.1

Syntax LRESULT CALLBACK ShellProc(code, wParam, lParam)

The **ShellProc** function is a library-defined callback function that a shell application can use to receive useful notifications from the system.

Parameters

code Specifies a shell-notification code. This parameter can be one of the following values:

Value	Meaning
HSHELL_ACTIVATESHELLWINDOW	The shell application should activate its main window.
HSHELL_WINDOWCREATED	A top-level, unowned window was created. The window exists when the system calls a **ShellProc** function.
HSHELL_WINDOWDESTROYED	A top-level, unowned window is about to be destroyed. The window still exists when the system calls a **ShellProc** function.

wParam Specifies additional information the shell application may need. The interpretation of this parameter depends on the value of the *code* parameter, as follows:

code	*wParam*
HSHELL_ACTIVATESHELLWINDOW	Not used.
HSHELL_WINDOWCREATED	Specifies the handle of the window being created.
HSHELL_WINDOWDESTROYED	Specifies the handle of the window being destroyed.

lParam Reserved; not used.

Return Value The return value should be zero.

Comments An application must install this callback function by specifying the WH_SHELL filter type and the procedure-instance address of the callback function in a call to the **SetWindowsHook** function.

ShellProc is a placeholder for the library-defined function name. The actual name must be exported by including it in an **EXPORTS** statement in the library's module-definition file.

See Also **DefHookProc, SendMessage, SetWindowsHook**

SpoolFile 3.1

Syntax HANDLE SpoolFile(lpszPrinter, lpszPort, lpszJob, lpszFile)

function SpoolFile(Printer, Port, Job, F: PChar): THandle;

The **SpoolFile** function puts a file into the spooler queue. This function is typically used by device drivers.

Parameters

lpszPrinter Points to a null-terminated string specifying the printer name—for example, "HP LasterJet IIP".

lpszPort Points to a null-terminated string specifying the local name—for example, "LPT1:". This must be a local port.

lpszJob Points to a null-terminated string specifying the name of the print job for the spooler. This string cannot be longer than 32 characters, including the null-terminating character.

lpszFile Points to a null-terminated string specifying the path and filename of the file to put in the spooler queue. This file contains raw printer data.

Return Value The return value is the global handle that is passed to the spooler, if the function is successful. Otherwise, it is an error value, which can be one of the following:

SP_APPABORT
SP_ERROR
SP_NOTREPORTED
SP_OUTOFDISK
SP_OUTOFMEMORY
SP_USERABORT

Comments Applications should ensure that the spooler is enabled before calling the **SpoolFile** function.

StackTraceCSIPFirst 3.1

Syntax #include <toolhelp.h>
BOOL StackTraceCSIPFirst(lpste, wSS, wCS, wIP, wBP)

function StackTraceCSIPFirst(lpStackTrace: PStackTraceEntry; wSS, wCS, wIP, wBP: Word): Bool;

The **StackTraceCSIPFirst** function fills the specified structure with information that describes the specified stack frame.

Parameters

lpste Points to a **STACKTRACEENTRY** structure to receive information about the stack. The **STACKTRACEENTRY** structure has the following form:

```
#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;
```

wSS Contains the value in the SS register. This value is used with the *wBP* value to determine the next entry in the stack trace.

wCS	Contains the value in the CS register of the first stack frame.
wIP	Contains the value in the IP register of the first stack frame.
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.

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

Comments The **StackTraceFirst** function can be used to begin a stack trace of any task except the current task. When a task is inactive, the kernel maintains its state, including its current stack, stack pointer, CS and IP values, and BP value. The kernel does not maintain these values for the current task. Therefore, when a stack trace is done on the current task, the application must use **StackTraceCSIPFirst** to begin a stack trace. An application can continue to trace through the stack by using the **StackTraceNext** function.

Before calling **StackTraceCSIPFirst**, an application must initialize the **STACKTRACEENTRY** structure and specify its size, in bytes, in the **dwSize** member.

See Also **StackTraceNext, StackTraceFirst**

StackTraceFirst 3.1

Syntax #include <toolhelp.h>
BOOL StackTraceFirst(lpste, htask)

function StackTraceFirst(lpStrackTrace: PStackTraceEntry; hTask: THandle): Bool;

The **StackTraceFirst** function fills the specified structure with information that describes the first stack frame for the given task.

Parameters *lpste* Points to a **STACKTRACEENTRY** structure to receive information about the task's first stack frame. The **STACKTRACEENTRY** structure has the following form:

```
#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;
```

htask Identifies the task whose stack information is to be described.

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

Comments The **StackTraceFirst** function can be used to begin a stack trace of any task except the current task. When a task is inactive, the kernel maintains its state, including its current stack, stack pointer, CS and IP values, and BP value. The kernel does not maintain these values for the current task. Therefore, when a stack trace is done on the current task, the application must use the **StackTraceCSIPFirst** function to begin a stack trace. An application can continue to trace through the stack by using the **StackTraceNext** function.

Before calling **StackTraceFirst**, an application must initialize the **STACKTRACEENTRY** structure and specify its size, in bytes, in the **dwSize** member.

See Also **StackTraceCSIPFirst, StackTraceNext**

StackTraceNext 3.1

Syntax #include <toolhelp.h>
BOOL StackTraceNext(lpste)

function StackTraceNext(lpStackTrace: PStackTraceEntry): Bool;

The **StackTraceNext** function fills the specified structure with information that describes the next stack frame in a stack trace.

Parameters *lpste* Points to a **STACKTRACEENTRY** structure to receive information about the next stack frame. The **STACKTRACEENTRY** structure has the following form:

```
#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;
```

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

Comments The **StackTraceNext** function can be used to continue a stack trace started by using the **StackTraceFirst** or **StackTraceCSIPFirst** function.

See Also **StackTraceCSIPFirst, StackTraceFirst, STACKTRACEENTRY**

StartDoc 3.1

Syntax int StartDoc(hdc, lpdi)

function StartDoc(DC: HDC; var di: TDocInfo): Integer;

The **StartDoc** function starts a print job. For Windows version 3.1, this function replaces the STARTDOC printer escape.

Parameters

hdc Identifies the device context for the print job.

lpdi Points to a **DOCINFO** structure containing the name of the document file and the name of the output file. The **DOCINFO** structure has the following form:

```
typedef struct {    /* di */
    int      cbSize;
    LPCSTR   lpszDocName;
    LPCSTR   lpszOutput;
} DOCINFO;
```

Return Value The return value is positive if the function is successful. Otherwise, it is SP_ERROR.

Comments Applications should call the **StartDoc** function immediately before beginning a print job. Using this function ensures that documents containing more than one page are not interspersed with other print jobs.

The **StartDoc** function should not be used inside metafiles.

See Also **EndDoc, Escape**

StartPage 3.1

Syntax int StartPage(hdc)

function StartPage(DC: HDC): Integer;

The **StartPage** function prepares the printer driver to accept data.

Parameters *hdc* Identifies the device context for the print job.

Return Value The return value is greater than zero if the function is successful. It is less than or equal to zero if an error occurs.

Comments The system disables the **ResetDC** function between calls to the **StartPage** and **EndPage** functions. This means that applications cannot change the device mode except at page boundaries.

See Also **EndPage, Escape, ResetDC**

SubtractRect 3.1

Syntax BOOL SubtractRect(lprcDest, lprcSource1, lprcSource2)

function SubtractRect(var lprcDest, lprcSource1, lprcSource2: TRect): Bool;

The **SubtractRect** function retrieves the coordinates of a rectangle by subtracting one rectangle from another.

Parameters

lprcDest Points to the **RECT** structure to receive the dimensions of the new rectangle. The **RECT** structure has the following form:

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

lprcSource1 Points to the **RECT** structure from which a rectangle is to be subtracted.

lprcSource2 Points to the **RECT** structure that is to be subtracted from the rectangle pointed to by the *lprcSource1* parameter.

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

Comments The rectangle specified by the *lprcSource2* parameter is subtracted from the rectangle specified by *lprcSource1* only when the rectangles intersect completely in either the x- or y-direction. For example, if *lprcSource1* were (10,10, 100,100) and *lprcSource2* were (50,50, 150,150), the rectangle pointed to by *lprcDest* would contain the same coordinates as *lprcSource1* when the function returned. If *lprcSource1* were (10,10, 100,100) and *lprcSource2* were (50,10, 150,150), however, the rectangle pointed to by *lprcDest* would contain the coordinates (10,10, 50,100) when the function returned.

See Also **IntersectRect, UnionRect**

SysMsgProc 3.1

Syntax LRESULT CALLBACK SysMsgProc(code, wParam, lParam)

The **SysMsgProc** function is a library-defined callback function that the system calls after a dialog box, message box, or menu has retrieved a message, but before the message is processed. The callback function can process or modify messages for any application in the system.

Parameters

code Specifies the type of message being processed. This parameter can be one of the following values:

Value	Meaning
MSGF_DIALOGBOX	Messages inside a dialog box or message box procedure are being processed.
MSGF_MENU	Keyboard and mouse messages in a menu are being processed.

If the *code* parameter is less than zero, the callback function must pass the message to the **CallNextHookEx** function without further processing and return the value returned by **CallNextHookEx**.

wParam Must be NULL.

lParam Points to the **MSG** structure to contain the message. The **MSG** structure has the following form:

```
typedef struct tagMSG {     /* msg */
    HWND   hwnd;
    UINT   message;
    WPARAM wParam;
    LPARAM lParam;
    DWORD  time;
```

```
    POINT  pt;
} MSG;
```

Return Value The return value should be nonzero if the function processes the message. Otherwise, it should be zero.

Comments This callback function must be in a dynamic-link library (DLL).

An application must install this callback function by specifying the WH_SYSMSGFILTER filter type and the procedure-instance address of the callback function in a call to the **SetWindowsHookEx** function.

SysMsgProc is a placeholder for the library-defined function name. The actual name must be exported by including it in an **EXPORTS** statement in the library's module-definition file.

See Also **CallNextHookEx, MessageBox, SetWindowsHookEx**

SystemHeapInfo 3.1

Syntax #include <toolhelp.h>
BOOL SystemHeapInfo(lpshi)

function SystemHeapInfo(lpSysHeap: PSysHeapInfo): Bool;

The **SystemHeapInfo** function fills the specified structure with information that describes the USER.EXE and GDI.EXE heaps.

Parameters *lpshi* Points to a **SYSHEAPINFO** structure to receive information about the USER and GDI heaps. The **SYSHEAPINFO** structure has the following form:

```
#include <toolhelp.h>

typedef struct tagSYSHEAPINFO {  /* shi */
    DWORD   dwSize;
    WORD    wUserFreePercent;
    WORD    wGDIFreePercent;
    HGLOBAL hUserSegment;
    HGLOBAL hGDISegment;
} SYSHEAPINFO;
```

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

Comments This function is included for advisory purposes. Before calling **SystemHeapInfo**, an application must initialize the **SYSHEAPINFO** structure and specify its size, in bytes, in the **dwSize** member.

SystemParametersInfo 3.1

Syntax BOOL SystemParametersInfo(uAction, uParam, lpvParam, fuWinIni)

function SystemParametersInfo(uAction, uParam: Word; lpvParam: Pointer; fuWinIni: Word): Bool;

The **SystemParametersInfo** function queries or sets system-wide parameters. This function can also update the WIN.INI file while setting a parameter.

Parameters *uAction* Specifies the system-wide parameter to query or set. This parameter can be one of the following values:

Value	Meaning
SPI_GETBEEP	Retrieves a **BOOL** value that indicates whether the warning beep is on or off.
SPI_GETBORDER	Retrieves the border multiplying factor that determines the width of a window's sizing border.
SPI_GETFASTTASKSWITCH	Determines whether fast task switching is on or off.
SPI_GETGRIDGRANULARITY	Retrieves the current granularity value of the desktop sizing grid.
SPI_GETICONTITLELOGFONT	Retrieves the logical-font information for the current icon-title font.
SPI_GETICONTITLEWRAP	Determines whether icon-title wrapping is on or off.
SPI_GETKEYBOARDDELAY	Retrieves the keyboard repeat-delay setting.
SPI_GETKEYBOARDSPEED	Retrieves the keyboard repeat-speed setting.
SPI_GETMENUDROPALIGNMENT	Determines whether pop-up menus are left-aligned or right-aligned relative to the corresponding menu-bar item.
SPI_GETMOUSE	Retrieves the mouse speed and the mouse threshold values, which Windows uses to calculate mouse acceleration.
SPI_GETSCREENSAVEACTIVE	Retrieves a **BOOL** value that indicates whether screen saving is on or off.
SPI_GETSCREENSAVETIMEOUT	Retrieves the screen-saver time-out value.
SPI_ICONHORIZONTALSPACING	Sets the width, in pixels, of an icon cell.
SPI_ICONVERTICALSPACING	Sets the height, in pixels, of an icon cell.

Value	Meaning
SPI_LANGDRIVER	Forces the user to load a new language driver.
SPI_SETBEEP	Turns the warning beep on or off.
SPI_SETBORDER	Sets the border multiplying factor that determines the width of a window's sizing border.
SPI_SETDESKPATTERN	Sets the current desktop pattern to the value specified in the Pattern entry in the WIN.INI file or to the pattern specified by the *lpvParam* parameter.
SPI_SETDESKWALLPAPER	Specifies the filename that contains the bitmap to be used as the desktop wallpaper.
SPI_SETDOUBLECLKHEIGHT	Sets the height of the rectangle within which the second click of a double-click must fall for it to be registered as a double-click.
SPI_SETDOUBLECLICKTIME	Sets the double-click time for the mouse. The double-click time is the maximum number of milliseconds that may occur between the first and second clicks of a double-click.
SPI_SETDOUBLECLKWIDTH	Sets the width of the rectangle within which the second click of a double-click must fall for it to be registered as a double-click.
SPI_SETFASTTASKSWITCH	Turns fast task switching on or off.
SPI_SETGRIDGRANULARITY	Sets the granularity of the desktop sizing grid.
SPI_SETICONTITLELOGFONT	Sets the font that is used for icon titles.
SPI_SETICONTITLEWRAP	Turns icon-title wrapping on or off.
SPI_SETKEYBOARDDELAY	Sets the keyboard repeat-delay setting.
SPI_SETKEYBOARDSPEED	Sets the keyboard repeat-speed setting.
SPI_SETMENUDROPALIGNMENT	Sets the alignment value of pop-up menus.
SPI_SETMOUSE	Sets the mouse speed and the x and y mouse-threshold values.
SPI_SETMOUSEBUTTONSWAP	Swaps or restores the meaning of the left and right mouse buttons.
SPI_SETSCREENSAVEACTIVE	Sets the state of the screen saver.
SPI_SETSCREENSAVETIMEOUT	Sets the screen-saver time-out value.

uParam	Depends on the *uAction* parameter. For more information, see the following Comments section.
lpvParam	Depends on the *uAction* parameter. For more information, see the following Comments section.
fuWinIni	If a system parameter is being set, specifies whether the WIN.INI file is updated, and if so, whether the WM_WININICHANGE message is broadcast to all top-level windows to notify them of the change. This parameter can be one of the following values:

Value	Meaning
SPIF_UPDATEINIFILE	Writes the new system-wide parameter setting to the WIN.INI file.
SPIF_SENDWININICHANGE	Broadcasts the WM_WININICHANGE message after updating the WIN.INI file.

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

Comments The **SystemParameterInfo** function is intended for applications, such as Control Panel, that allow the user to customize the Windows environment.

The following table describes the *uParam* and *lpvParam* parameters for each SPI_ constant:

Constant	*uParam*	*lpvParam*
SPI_GETBEEP	0	Points to a **BOOL** variable that receives TRUE if the beep is on, FALSE if it is off.
SPI_GETBORDER	0	Points to an integer variable that receives the border multiplying factor.
SPI_GETFASTTASKSWITCH	0	Points to a **BOOL** variable that receives TRUE if fast task switching is on, FALSE if it is off.
SPI_GETGRIDGRANULARITY	0	Points to an integer variable that receives the grid-granularity value.
SPI_GETICONTITLELOGFONT	Size of **LOGFONT** structure	Points to a **LOGFONT** structure that receives the logical-font information.
SPI_GETICONTITLEWRAP	0	Points to a **BOOL** variable that receives TRUE if wrapping is on, FALSE if wrapping is off.

Constant	*uParam*	*lpvParam*
SPI_GETKEYBOARDDELAY	0	Points to an integer variable that receives the keyboard repeat-delay setting.
SPI_GETKEYBOARDSPEED	0	Points to a **WORD** variable that receives the current keyboard repeat-speed setting.
SPI_GETMENUDROPALIGNMENT	0	Points to a **BOOL** variable that receives TRUE if pop-up menus are right-aligned, FALSE if they are left-aligned.
SPI_GETMOUSE	0	Points to an integer array name lpiMouse, where lpiMouse[0] receives the WIN.INI entry **MouseThreshold1**, lpiMouse[1] receives the entry **MouseThreshold2**, and lpiMouse[2] receives the entry **MouseSpeed**.
SPI_GETSCREENSAVEACTIVE	0	Points to a **BOOL** variable that receives TRUE if the screen saver is active, FALSE if it is not.
SPI_GETSCREENSAVETIMEOUT	0	Points to an integer variable that receives the screen-saver time-out value, in milliseconds.
SPI_ICONHORIZONTALSPACING	New width, in pixels, for horizontal spacing of icons	Is NULL if the icon cell width, in pixels, is returned in *uParam*. If this value is a pointer to an integer, the current horizontal spacing is returned in that variable and *uParam* is ignored.
SPI_ICONVERTICALSPACING	New height, in pixels, for vertical spacing of icons	Is NULL if the icon cell height, in pixels, is returned in *uParam*. If this value is a pointer to an integer, the current vertical spacing is returned in that variable and *uParam* is ignored.
SPI_LANGDRIVER	0	Points to a string containing the new language driver filename. The application should make sure that all other international settings remain consistent when changing the language driver.
SPI_SETBEEP	TRUE = turn the beep on; FALSE = turn the beep off	Is NULL.
SPI_SETBORDER	Border multiplying factor	Is NULL.

Constant	***uParam***	***lpvParam***
SPI_SETDESKPATTERN	0 or –1	Specifies the desktop pattern. If this value is NULL and the *uParam* parameter is –1, the value is reread from the WIN.INI file. This value can also be a null-terminated string (**LPSTR**) containing a sequence of 8 numbers that represent the new desktop pattern; for example, "170 85 170 85 170 85 170 85" represents a 50% gray pattern.
SPI_SETDESKWALLPAPER	0	Points to a string that specifies the name of the bitmap file.
SPI_SETDOUBLECLKHEIGHT	Double-click height, in pixels	Is NULL.
SPI_SETDOUBLECLICKTIME	Double-click time, in milliseconds	Is NULL.
SPI_SETDOUBLECLKWIDTH	Double-click width, in pixels	Is NULL.
SPI_SETFASTTASKSWITCH	TRUE = turn on fast task switching; FALSE = turn it off	Is NULL.
SPI_SETGRIDGRANULARITY	Grid granularity,	
SPI_SETICONTITLELOGFONT	Size of the **LOGFONT** structure	Points to a **LOGFONT** structure that defines the font to use for icon titles. If *uParam* is set to zero and *lParam* is set to NULL, Windows uses the icon-title font and spacings that were in effect when Windows was started.
SPI_SETICONTITLEWRAP	TRUE = turn wrapping on; FALSE = turn wrapping off	Is NULL.
SPI_SETKEYBOARDDELAY	Keyboard-delay setting	Is NULL.
SPI_SETKEYBOARDSPEED	Repeat-speed setting	Is NULL.
SPI_SETMENUDROPALIGNMENT	TRUE = right-alignment; FALSE = left-alignment	Is NULL.
SPI_SETMOUSE	0	Points to an integer array named lpiMouse, where lpiMouse[0] receives the WIN.INI entry **xMouseThreshold**, lpiMouse[1] receives the entry **yMouseThreshold**, and lpiMouse[2] receives the entry **MouseSpeed**.

Constant	*uParam*	*lpvParam*
SPI_SETMOUSEBUTTONSWAP	TRUE = reverse the meaning of the left and right mouse buttons; FALSE = restore the buttons to their original meanings	Is NULL.
SPI_SETSCREENSAVEACTIVE	TRUE = activate screen saving; FALSE = deactivate screen saving	Is NULL.
SPI_SETSCREENSAVETIMEOUT	Idle time-out duration, in seconds, before screen is saved	Is NULL.

Example The following example retrieves the value for the DoubleClickSpeed entry from the WIN.INI file and uses the value to initialize an edit control. In this example, while the WM_COMMAND message is being processed, the user-specified value is retrieved from the edit control and used to set the double-click time.

```
char szBuf[32];
int iResult;

case WM_INITDIALOG:

    /* Initialize edit control to the current double-click time. */

    iResult = GetProfileInt("windows",
        "DoubleClickSpeed", 550);
    itoa(iResult, szBuf, 10);
    SendDlgItemMessage(hdlg, IDD_DCLKTIME, WM_SETTEXT, 0,
        (DWORD) (LPSTR) szBuf);

    .
    . /* Initialize any other controls. */
    .

    return FALSE;

case WM_COMMAND:
    switch(wParam) {

        case IDOK:

            /* Set double-click time to a user-specified value. */

            SendDlgItemMessage(hdlg, IDD_DCLKTIME, WM_GETTEXT,
                sizeof(szBuf), (DWORD) (LPSTR) szBuf);
            SystemParametersInfo(SPI_SETDOUBLECLICKTIME, atoi(szBuf),
                (LPVOID) NULL, SPIF_UPDATEINIFILE |
                SPIF_SENDWININICHANGE);
```

```
        . /* Set any other system-wide parameters. */
        .

        EndDialog(hdlg, TRUE);
        return TRUE;
}
```

TaskFindHandle 3.1

Syntax #include <toolhelp.h>
BOOL TaskFindHandle(lpte, htask)

function TaskFindHandle(lpTask: PTaskEntry; hTask: THandle): Bool;

The **TaskFindHandle** function fills the specified structure with information that describes the given task.

Parameters

lpte Points to a **TASKENTRY** structure to receive information about the task. The **TASKENTRY** structure has the following form:

```
#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;
```

htask Identifies the task to be described.

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

Comments The **TaskFindHandle** function can be used to begin a walk through the task queue. An application can examine subsequent entries in the task queue by using the **TaskNext** function.

Before calling **TaskFindHandle**, an application must initialize the **TASKENTRY** structure and specify its size, in bytes, in the **dwSize** member.

See Also **TaskFirst, TaskNext**

TaskFirst 3.1

Syntax #include <toolhelp.h>
BOOL TaskFirst(lpte)

function TaskFirst(lpTask: PTaskEntry): Bool;

The **TaskFirst** function fills the specified structure with information about the first task on the task queue.

Parameters *lpte* Points to a **TASKENTRY** structure to receive information about the first task. The **TASKENTRY** structure has the following form:

```
#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;
```

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

Comments The **TaskFirst** function can be used to begin a walk through the task queue. An application can examine subsequent entries in the task queue by using the **TaskNext** function.

Before calling **TaskFirst**, an application must initialize the **TASKENTRY** structure and specify its size, in bytes, in the **dwSize** member.

See Also **TaskFindHandle, TaskNext**

TaskGetCSIP 3.1

Syntax #include <toolhelp.h>
DWORD TaskGetCSIP(htask)

function TaskGetCSIP(hTask: THandle): Longint;

The **TaskGetCSIP** function returns the next CS:IP value of a sleeping task. This function is useful for applications that must "know" where a sleeping task will begin execution upon awakening.

Parameters *htask* Identifies the task whose CS:IP value is being examined. This task must be sleeping when the application calls **TaskGetCSIP**.

Return Value The return value is the next CS:IP value, if the function is successful. If the *htask* parameter is invalid, the return value is NULL.

Comments **TaskGetCSIP** should not be called if *htask* identifies the current task.

See Also **DirectedYield, TaskSetCSIP, TaskSwitch**

TaskNext 3.1

Syntax #include <toolhelp.h>
BOOL TaskNext(lpte)

function TaskNext(lpTask: PTaskEntry): Bool;

The **TaskNext** function fills the specified structure with information about the next task on the task queue.

Parameters *lpte* Points to a **TASKENTRY** structure to receive information about the next task. The **TASKENTRY** structure has the following form:

```
#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;
```

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

Comments The **TaskNext** function can be used to continue a walk through the task queue. The walk must have been started by the **TaskFirst** or **TaskFindHandle** function.

See Also **TaskFindHandle, TaskFirst**

TaskSetCSIP 3.1

Syntax #include <toolhelp.h>
DWORD TaskSetCSIP(htask, wCS, wIP)

function TaskSetCSIP(hTask: THandle; wCS, wIP: Word): Longint;

The **TaskSetCSIP** function sets the CS:IP value of a sleeping task. When the task is yielded to, it will begin execution at the specified address.

Parameters

htask	Identifies the task to be assigned the new CS:IP value.
wCS	Contains the new value of the CS register.
wIP	Contains the new value of the IP register.

Return Value The return value is the previous CS:IP value for the task. The **TaskSwitch** function uses this value. The return value is NULL if the *htask* parameter is invalid.

Comments **TaskSetCSIP** should not be called if *htask* identifies the current task.

See Also **DirectedYield, TaskGetCSIP, TaskSwitch**

TaskSwitch 3.1

Syntax #include <toolhelp.h>
BOOL TaskSwitch(htask, dwNewCSIP)

function TaskSwitch(hTask: THandle; dwNewCSIP: Longint): Bool;

The **TaskSwitch** function switches to the given task. The task begins executing at the specified address.

Parameters

htask	Identifies the new task.
dwNewCSIP	Identifies the address within the given task at which to begin execution. Be very careful that this address is not in a code segment owned by the given task.

Return Value The return value is nonzero if the task switch is successful. Otherwise, it is zero.

Comments When the task identified by the *htask* parameter yields, **TaskSwitch** returns to the calling application.

TaskSwitch changes the CS:IP value of the task's stack frame to the value specified by the *dwNewCSIP* parameter and then calls the **DirectedYield** function.

See Also **DirectedYield, TaskSetCSIP, TaskGetCSIP**

TerminateApp 3.1

Syntax #include <toolhelp.h>
void TerminateApp(htask, wFlags)

procedure TerminateApp(hTask: THandle; wFlags: Word);

The **TerminateApp** function ends the given application instance (task).

Parameters

htask	Identifies the task to be ended. If this parameter is NULL, it identifies the current task.

wFlags Indicates how to end the task. This parameter can be one of the following values:

Value	Meaning
UAE_BOX	Calls the Windows kernel to display the Application Error message box and then ends the task.
NO_UAE_BOX	Calls the Windows kernel to end the task but does not display the Application Error message box. The application's interrupt or notification callback function should have displayed an error message, a warning, or both.

Return Value This function returns only if *htask* is not NULL and does not identify the current task.

Comments The **TerminateApp** function unregisters all callback functions registered with the Tool Help functions and then ends the application as if the given task had produced a general-protection (GP) fault or other error.

TerminateApp should be used only by debugging applications, because the function may not free not all objects owned by the ended application.

See Also **InterruptRegister, InterruptUnRegister, NotifyRegister, NotifyUnRegister**

TimerCount 3.1

Syntax #include <toolhelp.h>
BOOL TimerCount(lpti)

function TimerCount(lpTimer: PTimerInfo): Bool;

The **TimerCount** function fills the specified structure with the execution times of the current task and VM (virtual machine).

Parameters *lpti* Points to the **TIMERINFO** structure that will receive the execution times. The **TIMERINFO** structure has the following form:

```
#include <toolhelp.h>

typedef struct tagTIMERINFO {  /* ti */
    DWORD dwSize;
    DWORD dwmsSinceStart;
```

```
    DWORD dwmsThisVM;
} TIMERINFO;
```

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

Comments The **TimerCount** function provides a consistent source of timing information, accurate to the millisecond. In enhanced mode, **TimerCount** uses the VTD (virtual timer device) to obtain accurate execution times.

In standard mode, **TimerCount** calls the **GetTickCount** function, which returns information accurate to one clock tick (approximately 55 ms). **TimerCount** then reads the hardware timer to estimate how many milliseconds remain until the next clock tick. The resulting time is accurate to 1 ms.

Before calling **TimerCount**, an application must initialize the **TIMERINFO** structure and specify its size, in bytes, in the **dwSize** member.

See Also **GetTickCount**

TimerProc 2.x

Syntax void CALLBACK TimerProc(hwnd, msg, idTimer, dwTime)

The **TimerProc** function is an application-defined callback function that processes WM_TIMER messages.

Parameters

hwnd	Identifies the window associated with the timer.
msg	Specifies the WM_TIMER message.
idTimer	Specifies the timer's identifier.
dwTime	Specifies the current system time.

Return Value This function does not return a value.

Comments **TimerProc** is a placeholder for the application-defined function name. The actual name must be exported by including it in an **EXPORTS** statement in the application's module-definition file.

See Also **KillTimer, SetTimer**

UnAllocDiskSpace 3.1

Syntax #include <stress.h>
void UnAllocDiskSpace(drive)

procedure UnAllocDiskSpace(wDrive: Word);

The **UnAllocDiskSpace** function deletes the STRESS.EAT file from the root directory of the specified drive. This frees the disk space previously consumed by the **AllocDiskSpace** function.

Parameters *drive* Specifies the disk partition on which to delete the STRESS.EAT file. This can be one of the following values:

Value	Meaning
EDS_WIN	Deletes the file on the Windows partition.
EDS_CUR	Deletes the file on the current partition.
EDS_TEMP	Deletes the file on the partition that contains the TEMP directory.

Return Value This function does not return a value.

See Also **AllocDiskSpace**

UnAllocFileHandles 3.1

Syntax #include <stress.h>
void UnAllocFileHandles(void)

procedure UnAllocFileHandles;

The **UnAllocFileHandles** function frees all file handles allocated by the **AllocFileHandles** function.

Parameters This function has no parameters.

Return Value This function does not return a value.

See Also **AllocFileHandles**

UndeleteFile

Syntax #include <wfext.h>
int FAR PASCAL UndeleteFile(hwndParent, lpszDir)

TFM_UnDelete_Proc = function(Handle: HWnd; P: PChar): Longint;

The **UndeleteFile** function is an application-defined callback function that File Manager calls when the user chooses the Undelete command from the File Manager File menu.

Parameters

hwndParent Identifies the File Manager window. An "undelete" dynamic-link library (DLL) should use this handle to specify the parent window for any dialog box or message box the DLL may display.

lpszDir Points to a null-terminated string that contains the name of the initial directory.

Return Value The return value is one of the following, if the function is successful:

Value	Meaning
–1	An error occurred.
IDOK	A file was undeleted. File Manager will repaint its windows.
IDCANCEL	No file was undeleted.

UnhookWindowsHookEx 3.1

Syntax BOOL UnhookWindowsHookEx(hhook)

function UnhookWindowsHookEx(Hook: HHook): Bool;

The **UnhookWindowsHookEx** function removes an application-defined hook function from a chain of hook functions. A hook function processes events before they are sent to an application's message loop in the **WinMain** function.

Parameters

hhook Identifies the hook function to be removed. This is the value returned by the **SetWindowsHookEx** function when the hook was installed.

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

Comments The **UnhookWindowsHookEx** function must be used in combination with the **SetWindowsHookEx** function.

Example The following example uses the **UnhookWindowsHookEx** function to remove a message filter that was used to provide context-sensitive help for a dialog box:

```
DLGPROC lpfnAboutProc;
HOOKPROC lpfnFilterProc;
HHOOK   hhook;

caseIDM_ABOUT:
    lpfnAboutProc = (DLGPROC) MakeProcInstance(About, hinst);
    lpfnFilterProc = (HOOKPROC) MakeProcInstance(FilterFunc, hinst);
    hhook = SetWindowsHookEx(WH_MSGFILTER, lpfnFilterProc,
        hinst, (HTASK) NULL);

    DialogBox(hinst, "AboutBox", hwnd, lpfnAboutProc);

    UnhookWindowsHookEx(hhook);
    FreeProcInstance((FARPROC) lpfnFilterProc);
    FreeProcInstance((FARPROC) lpfnAboutProc);

    break;
```

See Also **CallNextHookEx, SetWindowsHookEx**

VerFindFile 3.1

Syntax #include <ver.h>
UINT VerFindFile(flags, lpszFilename, lpszWinDir, lpszAppDir, lpszCurDir, lpuCurDirLen, lpszDestDir, lpuDestDirLen)

function VerFindFile(Flags: Word; FileName, WinDir, AppDir, CurDir: PChar; var CurDirLen: Word; DestDir: PChar; var DestDirLen: Word): Word;

The **VerFindFile** function determines where to install a file based on whether it locates another version of the file in the system. The values **VerFindFile** returns are used in a subsequent call to the **VerInstallFile** function.

Parameters

flags	Contains a bitmask of flags. This parameter can be VFFF_ISSHAREDFILE, which indicates that the source file may be shared by multiple applications. **VerFindFile** uses this information to determine where the file should be copied. All other values are reserved for future use.

lpszFilename	Points to a null-terminated string specifying the name of the file to be installed. This name should include only the filename and extension, not a path.
lpszWinDir	Points to a null-terminated string specifying the Windows directory. This string is returned by the **GetWindowsDir** function. The dynamic-link library (DLL) version of **VerFindFile** ignores this parameter.
lpszAppDir	Points to a null-terminated string specifying the drive letter and directory where the installation program is installing a set of related files. If the installation program is installing an application, this is the directory where the application will reside. This directory will also be the application's working directory unless you specify otherwise.
lpszCurDir	Points to a buffer that receives the path to a current version of the file being installed. The path is a null-terminated string. If a current version is not installed, the buffer will contain the source directory of the file being installed. The buffer must be at least _MAX_PATH bytes long.
lpuCurDirLen	Points to a null-terminated string specifying the length, in bytes, of the buffer pointed to by *lpszCurDir*. On return, *lpuCurDirLen* contains the size, in bytes, of the data returned in *lpszCurDir*, including the terminating null character. If the buffer is too small to contain all the data, *lpuCurDirLen* will be greater than the actual size of the buffer.
lpszDestDir	Points to a buffer that receives the path to the installation directory recommended by **VerFindFile**. The path is a null-terminated string. The buffer must be at least _MAX_PATH bytes long.
lpuDestDirLen	Points to the length, in bytes, of the buffer pointed to by *lpszDestDir*. On return, *lpuDestDirLen* contains the size, in bytes, of the data returned in *lpszDestDir*, including the terminating null character. If the buffer is too small to contain all the data, *lpuDestDirLen* will be greater than the actual size of the buffer.

Return Value The return value is a bitmask that indicates the status of the file, if the function is successful. This value may be one or more of the following:

Error	Meaning
VFF_CURNEDEST	Indicates that the currently installed version of the file is not in the recommended destination.
VFF_FILEINUSE	Indicates that Windows is using the currently installed version of the file; therefore, the file cannot be overwritten or deleted.
VFF_BUFFTOOSMALL	Indicates that at least one of the buffers was too small to contain the corresponding string. An application should check the *lpuCurDirLen* and *lpuDestDirLen* parameters to determine which buffer was too small.

All other values are reserved for future use.

Comments The dynamic-link library (DLL) version of **VerFindFile** searches for a copy of the specified file by using the **OpenFile** function. In the LIB version, the function searches for the file in the Windows directory, the system directory, and then the directories specified by the PATH environment variable.

VerFindFile determines the system directory from the specified Windows directory, or it searches the path.

If the *flags* parameter indicates that the file is private to this application (not VFFF_ISSHAREDFILE), **VerFindFile** recommends installing the file in the application's directory. Otherwise, if the system is running a shared copy of Windows, the function recommends installing the file in the Windows directory. If the system is running a private copy of Windows, the function recommends installing the file in the system directory.

See Also **VerInstallFile**

VerInstallFile 3.1

Syntax #include <ver.h>
DWORD VerInstallFile(flags, lpszSrcFilename, lpszDestFilename, lpszSrcDir, lpszDestDir, lpszCurDir, lpszTmpFile, lpwTmpFileLen)

function VerInstallFile(Flags: Word; SrcFileName, DestFileName, SrcDir, DestDir, CurDir, TmpFile: PChar; var TmpFileLen: Word): Longint;

The **VerInstallFile** function attempts to install a file based on information returned from the **VerFindFile** function. **VerInstallFile** decompresses the file with the **LZCopy** function and checks for errors, such as outdated files.

Parameters *flags* Contains a bitmask of flags. This parameter can be a combination of the following values:

Value	Meaning
VIFF_FORCEINSTALL	Installs the file regardless of mismatched version numbers. The function will check only for physical errors during installation. If *flags* includes VIFF_FORCEINSTALL and *lpszTmpFileLen* is not a pointer to zero, **VerInstallFile** will skip all version checks of the temporary file and the destination file and rename the temporary file to the name specified by *lpszSrcFilename*, as long as the temporary file exists in the destination directory, the destination file is not in use, and the user has privileges to delete the destination file and rename the temporary file. The return value from **VerInstallFile** should be checked for any errors.
VIFF_DONTDELETEOLD	Installs the file without deleting the previously installed file, if the previously installed file is not in the destination directory. If the previously installed file is in the destination directory, **VerInstallFile** replaces it with the new file upon successful installation.

	All other values are reserved for future use.
lpszSrcFilename	Points to the name of the file to be installed. This is the filename in the directory pointed to by *lpszSrcDir*; the filename should include only the filename and extension, not a path. **VerInstallFile** opens the source file by using the **LZOpenFile** function. This means it can handle both files as specified and files that have been compressed and renamed by using the /r option with COMPRESS.EXE.
lpszDestFilename	Points to the name **VerInstallFile** will give the new file upon installation. This filename may be different than the filename in the directory pointed to by *lpszSrcFilename*. The new name should include only the filename and extension, not a path.
lpszSrcDir	Points to a buffer that contains the directory name where the new file is found.
lpszDestDir	Points to a buffer that contains the directory name where the new file should be installed. The **VerFindFile** function returns this value in the *lpszDestDir* parameter.
lpszCurDir	Points to a buffer that contains the directory name where the preexisting version of this file is found. **VerFindFile** returns this value in the *lpszCurDir* parameter. If the filename specified in *lpszDestFilename* already exists in the *lpszCurDir* directory and *flags* does not include VIFF_DONTDELETEOLD, the existing file will be deleted. If *lpszCurDir* is a pointer to NULL, a previous version of the file does not exist on the system.
lpszTmpFile	Points to a buffer that should be empty upon the initial call to **VerInstallFile**. The function fills the buffer with the name of a temporary copy of the source file. The buffer must be at least _MAX_PATH bytes long.
lpwTmpFileLen	Points to the length of the buffer pointed to by *lpszTmpFile*. On return, *lpwTmpFileLen* contains the size, in bytes, of the data returned in *lpszTmpFile*, including the terminating null character. If the buffer is too small to contain all the data, *lpwTmpFileLen* will be greater than the actual size of the buffer.

If *flags* includes VIFF_FORCEINSTALL and *lpwTmpFileLen* is not a pointer to zero, **VerInstallFile** will rename the temporary file to the name specified by *lpszSrcFilename.*

Return Value The return value is a bitmask that indicates exceptions, if the function is successful. This value may be one or more of the following:

Value	Meaning
VIF_TEMPFILE	Indicates that the temporary copy of the new file is in the destination directory. The cause of failure is reflected in other flags. Applications should always check whether this bit is set and delete the temporary file, if required.
VIF_MISMATCH	Indicates that the new and preexisting files differ in one or more attributes. This error can be overridden by calling **VerInstallFile** again with the VIFF_FORCEINSTALL flag.
VIF_SRCOLD	Indicates that the file to install is older than the preexisting file. This error can be overridden by calling **VerInstallFile** again with the VIFF_FORCEINSTALL flag.
VIF_DIFFLANG	Indicates that the new and preexisting files have different language or code-page values. This error can be overridden by calling **VerInstallFile** again with the VIFF_FORCEINSTALL flag.
VIF_DIFFCODEPG	Indicates that the new file requires a code page that cannot be displayed by the currently running version of Windows. This error can be overridden by calling **VerInstallFile** with the VIFF_FORCEINSTALL flag.
VIF_DIFFTYPE	Indicates that the new file has a different type, subtype, or operating system than the preexisting file. This error can be overridden by calling **VerInstallFile** again with the VIFF_FORCEINSTALL flag.
VIF_WRITEPROT	Indicates that the preexisting file is write-protected. The installation program should reset the read-only bit in the destination file before proceeding with the installation.
VIF_FILEINUSE	Indicates that the preexisting file is in use by Windows and cannot be deleted.
VIF_OUTOFSPACE	Indicates that the function cannot create the temporary file due to insufficient disk space on the destination drive.
VIF_ACCESSVIOLATION	Indicates that a create, delete, or rename operation failed due to an access violation.

Value	Meaning
VIF_SHARINGVIOLATION	Indicates that a create, delete, or rename operation failed due to a sharing violation.
VIF_CANNOTCREATE	Indicates that the function cannot create the temporary file. The specific error may be described by another flag.
VIF_CANNOTDELETE	Indicates that the function cannot delete the destination file or cannot delete the existing version of the file located in another directory. If the VIF_TEMPFILE bit is set, the installation failed and the destination file probably cannot be deleted.
VIF_CANNOTRENAME	Indicates that the function cannot rename the temporary file but already deleted the destination file.
VIF_OUTOFMEMORY	Indicates that the function cannot complete the requested operation due to insufficient memory. Generally, this means the application ran out of memory attempting to expand a compressed file.
VIF_CANNOTREADSRC	Indicates that the function cannot read the source file. This could mean that the path was not specified properly, that the file does not exist, or that the file is a compressed file that has been corrupted. To distinguish these conditions, use **LZOpenFile** to determine whether the file exists. (Do not use the **OpenFile** function, because it does not correctly translate filenames of compressed files.) Note that VIF_CANNOTREADSRC does not cause either the VIF_ACCESSVIOLATION or VIF_SHARINGVIOLATION bit to be set.
VIF_CANNOTREADDST	Indicates that the function cannot read the destination (existing) files. This prevents the function from examining the file's attributes.
VIF_BUFFTOOSMALL	Indicates that the *lpszTmpFile* buffer was too small to contain the name of the temporary source file. On return, *lpwTmpFileLen* contains the size of the buffer required to hold the filename.

All other values are reserved for future use.

Comments **VerInstallFile** is designed for use in an installation program. This function copies a file (specified by *lpszSrcFilename*) from the installation disk to a temporary file in the destination directory. If necessary, **VerInstallFile** expands the file by using the functions in LZEXPAND.DLL.

If a preexisting copy of the file exists in the destination directory, **VerInstallFile** compares the version information of the temporary file to that of the preexisting file. If the preexisting file is more recent than the new version, or if the files' attributes are significantly different, **VerInstallFile** returns one or more error values. For example, files with different languages would cause **VerInstallFile** to return VIF_DIFFLANG.

VerInstallFile leaves the temporary file in the destination directory. If all of the errors are recoverable, the installation program can override them by calling **VerInstallFile** again with the VIFF_FORCEINSTALL flag. In this case, *lpszSrcFilename* should point to the name of the temporary file. Then, **VerInstallFile** deletes the preexisting file and renames the temporary file to the name specified by *lpszSrcFilename*. If the VIF_TEMPFILE bit indicates that a temporary file exists and the application does not force the installation by using the VIFF_FORCEINSTALL flag, the application must delete the temporary file.

If an installation program attempts to force installation after a nonrecoverable error, such as VIF_CANNOTREADSRC, **VerInstallFile** will not install the file.

See Also **VerFindFile**

VerLanguageName 3.1

Syntax #include <ver.h>
UINT VerLanguageName(uLang, lpszLang, cbLang)

function VerLanguageName(Lang:Word; Lang: PChar; Size: Word): Word;

The **VerLanguageName** function converts the specified binary Microsoft language identifier into a text representation of the language.

Parameters

uLang	Specifies the binary Microsoft language identifier. For example, **VerLanguageName** translates 0x040A into Castilian Spanish. If **VerLanguageName** does not recognize the identifier, the *lpszLang* parameter will point to a default string, such as "Unknown language". For a complete list of the language identifiers supported by Windows, see the following Comments section.
lpszLang	Points to the buffer to receive the null-terminated string representing the language specified by the *uLang* parameter.

cbLang Indicates the size of the buffer, in bytes, pointed to by *lpszLang*.

Return Value The return value is the length of the string that represents the language identifier, if the function is successful. This value does not include the null character at the end of the string. If this value is greater than *cbLang*, the string was truncated to *cbLang*. The return value is zero if an error occurs. Unknown *uLang* values do not produce errors.

Comments Typically, an installation application uses this function to translate a language identifier returned by the **VerQueryValue** function. The text string may be used in a dialog box that asks the user how to proceed in the event of a language conflict.

Windows supports the following language identifiers:

Value	Language
0x0401	Arabic
0x0402	Bulgarian
0x0403	Catalan
0x0404	Traditional Chinese
0x0405	Czech
0x0406	Danish
0x0407	German
0x0408	Greek
0x0409	U.S. English
0x040A	Castilian Spanish
0x040B	Finnish
0x040C	French
0x040D	Hebrew
0x040E	Hungarian
0x040F	Icelandic
0x0410	Italian
0x0411	Japanese
0x0412	Korean
0x0413	Dutch
0x0414	Norwegian – Bokmål
0x0415	Polish
0x0416	Brazilian Portuguese
0x0417	Rhaeto-Romanic
0x0418	Romanian
0x0419	Russian
0x041A	Croato-Serbian (Latin)

Value	Language
0x041B	Slovak
0x041C	Albanian
0x041D	Swedish
0x041E	Thai
0x041F	Turkish
0x0420	Urdu
0x0421	Bahasa
0x0804	Simplified Chinese
0x0807	Swiss German
0x0809	U.K. English
0x080A	Mexican Spanish
0x080C	Belgian French
0x0810	Swiss Italian
0x0813	Belgian Dutch
0x0814	Norwegian – Nynorsk
0x0816	Portuguese
0x081A	Serbo-Croatian (Cyrillic)
0x0C0C	Canadian French
0x100C	Swiss French

VerQueryValue 3.1

Syntax

```
#include <ver.h>
BOOL VerQueryValue(lpvBlock, lpszSubBlock, lplpBuffer, lpcb)
```

```
function VerQueryValue(Block: Pointer; SubBlock: PChar; var Buffer:
Pointer; var Len: Word): Bool;
```

The **VerQueryValue** function returns selected version information from the specified version-information resource. To obtain the appropriate resource, the **GetFileVersionInfo** function must be called before **VerQueryValue**.

Parameters

lpvBlock — Points to the buffer containing the version-information resource returned by the **GetFileVersionInfo** function.

lpszSubBlock — Points to a zero-terminated string specifying which version-information value to retrieve. The string consists of names separated by backslashes (\ ) and can have one of the following forms:

Form	Description
\	Specifies the root block. The function retrieves a pointer to the **VS_FIXEDFILEINFO** structure for the version-information resource.
\VarFileInfo\Translation	Specifies the translation table in the variable information block. The function retrieves a pointer to an array of language and character-set identifiers. An application uses these identifiers to create the name of an language-specific block in the version-information resource.
\StringFileInfo_lang-charset__string-name_	Specifies a value in a language-specific block. The *lang-charset* name is a concatenation of a language and character-set identifier pair found in the translation table for the resource. The *lang-charset* name must be specified as a hexadecimal string. The *string-name* name is one of the predefined strings described in the following Comments section.

lplpBuffer Points to a buffer that receives a pointer to the version-information value.

lpcb Points to a buffer that receives the length, in bytes, of the version-information value.

Return Value The return value is nonzero if the specified block exists and version information is available. If *lpcb* is zero, no value is available for the specified version-information name. The return value is zero if the specified name does not exist or the resource pointed to by *lpvBlock* is not valid.

Comments The *string-name* in the *lpszSubBlock* parameter can be one of the following predefined names:

Name	Value
Comments	Specifies additional information that should be displayed for diagnostic purposes.
CompanyName	Specifies the company that produced the file—for example, "Microsoft Corporation" or "Standard Microsystems Corporation, Inc.". This string is required.

Name	Value
FileDescription	Specifies a file description to be presented to users. This string may be displayed in a list box when the user is choosing files to install—for example, "Keyboard Driver for AT-Style Keyboards" or "Microsoft Word for Windows". This string is required.
FileVersion	Specifies the version number of the file—for example, "3.10" or "5.00.RC2". This string is required.
InternalName	Specifies the internal name of the file, if one exists—for example, a module name if the file is a dynamic-link library. If the file has no internal name, this string should be the original filename, without extension. This string is required.
LegalCopyright	Specifies all copyright notices that apply to the file. This should include the full text of all notices, legal symbols, copyright dates, and so on—for example, "Copyright Microsoft Corporation 1990–1991". This string is optional.
LegalTrademarks	Specifies all trademarks and registered trademarks that apply to the file. This should include the full text of all notices, legal symbols, trademark numbers, and so on—for example, "Windows(TM) is a trademark of Microsoft Corporation". This string is optional.
OriginalFilename	Specifies the original name of the file, not including a path. This information enables an application to determine whether a file has been renamed by a user. The format of the name depends on the file system for which the file was created. This string is required.
PrivateBuild	Specifies information about a private version of the file—for example, "Built by TESTER1 on \TESTBED". This string should be present only if the VS_FF_PRIVATEBUILD flag is set in the **dwFileFlags** member of the **VS_FIXEDFILEINFO** structure of the root block.
ProductName	Specifies the name of the product with which the file is distributed—for example, "Microsoft Windows". This string is required.
ProductVersion	Specifies the version of the product with which the file is distributed—for example, "3.10" or "5.00.RC2". This string is required.
SpecialBuild	Specifies how this version of the file differs from the standard version—for example, "Private build for TESTER1 solving mouse problems on M250 and M250E computers". This string should be present only if the VS_FF_SPECIALBUILD flag is set in the dwFileFlags member of the **VS_FIXEDFILEINFO** structure in the root block.

Example The following example loads the version information for a dynamic-link library and retrieves the company name:

```
BYTE   abData[512];
DWORD  handle;
DWORD  dwSize;
LPBYTE lpBuffer;
char   szName[512];

dwSize = GetFileVersionInfoSize("c:\\dll\\sample.dll", &handle));

GetFileVersionInfo("c:\\dll\\sample.dll", handle, dwSize, abData));

VerQueryValue(abData, "\\VarFileInfo\\Translation", &lpBuffer,
&dwSize));

if (dwSize!=0) {
    wsprintf(szName, "\\StringFileInfo\\%8lx\\CompanyName", &lpBuffer);
    VerQueryValue(abData, szName, &lpBuffer, &dwSize);
}
```

See Also **GetFileVersionInfo**

WindowProc 2.x

Syntax LRESULT CALLBACK WindowProc(hwnd, msg, wParam, lParam)

The **WindowProc** function is an application-defined callback function that processes messages sent to a window.

Parameters

hwnd	Identifies the window.
msg	Specifies the message.
wParam	Specifies 16 bits of additional message-dependent information.
lParam	Specifies 32 bits of additional message-dependent information.

Return Value The return value is the result of the message processing. The value depends on the message being processed.

Comments The **WindowProc** name is a placeholder for the application-defined function name. The actual name must be exported by including it in an **EXPORTS** statement in the application's module-definition file.

See Also **DefWindowProc, RegisterClass**

WNetAddConnection 3.1

Syntax UINT WNetAddConnection(lpszNetPath, lpszPassword, lpszLocalName)

function WNetAddConnection(lpszNetPath, lpszPassword,
lpszLocalName: PChar): Word;

The **WNetAddConnection** function redirects the specified local device (either a disk drive or a printer port) to the given shared device or remote server.

Parameters

lpszNetPath Points to a null-terminated string specifying the shared device or remote server.

lpszPassword Points to a null-terminated string specifying the network password for the given device or server.

lpszLocalName Points to a null-terminated string specifying the local drive or device to be redirected. All *lpszLocalName* strings (such as LPT1) are case-independent. Only the drive names A through Z and the device names LPT1 through LPT3 are used.

Return Value The return value is one of the following:

Value	Meaning
WN_SUCCESS	The function was successful.
WN_NOT_SUPPORTED	The function was not supported.
WN_OUT_OF_MEMORY	The system was out of memory.
WN_NET_ERROR	An error occurred on the network.
WN_BAD_POINTER	The pointer was invalid.
WN_BAD_NETNAME	The network resource name was invalid.
WN_BAD_LOCALNAME	The local device name was invalid.
WN_BAD_PASSWORD	The password was invalid.
WN_ACCESS_DENIED	A security violation occurred.
WN_ALREADY_CONNECTED	The local device was already connected to a remote resource.

See Also **WNetCancelConnection, WNetGetConnection**

WNetCancelConnection 3.1

Syntax UINT WNetCancelConnection(lpszName, fForce)

function WNetCancelConnection(lpszName: PChar; tForce: Bool): Word;

The **WNetCancelConnection** function cancels a network connection.

Parameters

lpszName Points to either the name of the redirected local device (such as LPT1) or a fully qualified network path. If a network path is specified, the driver cancels all the connections to that resource.

fForce Specifies whether any open files or open print jobs on the device should be closed before the connection is canceled. If this parameter is FALSE and there are open files or jobs, the connection should not be canceled and the function should return the WN_OPEN_FILES error value.

Return Value The return value is one of the following:

Value	Meaning
WN_SUCCESS	The function was successful.
WN_NOT_SUPPORTED	The function was not supported.
WN_OUT_OF_MEMORY	The system was out of memory.
WN_NET_ERROR	An error occurred on the network.
WN_BAD_POINTER	The pointer was invalid.
WN_BAD_VALUE	The *lpszName* parameter was not a valid local device or network name.
WN_NOT_CONNECTED	The *lpszName* parameter was not a redirected local device or currently accessed network resource.
WN_OPEN_FILES	Files were open and the *fForce* parameter was FALSE. The connection was not canceled.

See Also **WNetAddConnection, WNetGetConnection**

WNetGetConnection 3.1

Syntax UINT WNetGetConnection(lpszLocalName, lpszRemoteName, cbRemoteName)

function WNetGetConnection(lpszLocalName, lpszRemoteName: PChar; cbBufferSize: PWord): Word;

The **WNetGetConnection** function returns the name of the network resource associated with the specified redirected local device.

Parameters

lpszLocalName	Points to a null-terminated string specifying the name of the redirected local device.
lpszRemoteName	Points to the buffer to receive the null-terminated name of the remote network resource.
cbRemoteName	Points to a variable specifying the maximum number of bytes the buffer pointed to by *lpszRemoteName* can hold. The function sets this variable to the number of bytes copied to the buffer.

Return Value The return value is one of the following:

Value	Meaning
WN_SUCCESS	The function was successful.
WN_NOT_SUPPORTED	The function was not supported.
WN_OUT_OF_MEMORY	The system was out of memory.
WN_NET_ERROR	An error occurred on the network.
WN_BAD_POINTER	The pointer was invalid.
WN_BAD_VALUE	The *szLocalName* parameter was not a valid local device.
WN_NOT_CONNECTED	The *szLocalName* parameter was not a redirected local device.
WN_MORE_DATA	The buffer was too small.

See Also **WNetAddConnection, WNetCancelConnection**

WordBreakProc 3.1

Syntax int CALLBACK WordBreakProc(lpszEditText, ichCurrentWord, cbEditText, action)

TEditWordBreakProc = function(lpch: PChar; ichCurrent: Integer; cch: Integer; Code: Integer): Integer;

The **WordBreakProc** function is an application-defined callback function that the system calls whenever a line of text in a multiline edit control must be broken.

Parameters

lpszEditText	Points to the text of the edit control.

ichCurrentWord	Specifies an index to a word in the buffer of text that identifies the point at which the function should begin checking for a word break.
cbEditText	Specifies the number of bytes in the text.
action	Specifies the action to be taken by the callback function. This parameter can be one of the following values:

Value	Action
WB_LEFT	Look for the beginning of a word to the left of the current position.
WB_RIGHT	Look for the beginning of a word to the right of the current position.
WB_ISDELIMITER	Check whether the character at the current position is a delimiter.

Return Value If the *action* parameter specifies WB_ISDELIMITER, the return value is non-zero (TRUE) if the character at the current position is a delimiter, or zero if it is not. Otherwise, the return value is an index to the begining of a word in the buffer of text.

Comments A carriage return (CR) followed by a linefeed (LF) must be treated as a single word by the callback function. Two carriage returns followed by a linefeed also must be treated as a single word.

An application must install the callback function by specifying the procedure-instance address of the callback function in a EM_SETWORDBREAKPROC message.

WordBreakProc is a placeholder for the library-defined function name. The actual name must be exported by including it in an **EXPORTS** statement in the library's module-definition file.

See Also **SendMessage**

C H A P T E R

5

Data types

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.

Type	Definition
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.
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.

Type	Definition
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.
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.

Type	Definition
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.
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.

Type	Definition
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.
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.

Type	Definition
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.

Type	Definition
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.
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.

Type	Definition
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.
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.

C H A P T E R

6

Messages

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:

```
DWORDdwIndex;

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_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 was created with an owner-drawn style but without the CBS_HASSTRINGEXS style, this message returns the index of the item whose doubleword value matches the value of the *lpszFind* parameter.

See Also CB_FINDSTRING, 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:

```
BOOLfDropped;

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

See Also CB_SHOWDROPDOWN

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_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_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_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:

```
LPARAMlrHeight;

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

See Also CB_GETITEMHEIGHT

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_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_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_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_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 line on the display.

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**

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, this message returns the index of the item whose doubleword value (supplied for the *lParam* parameter of the LB_ADDSTRING or LB_INSERTSTRING message) matches the value supplied for the *lpszFind*
parameter.

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_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:

```
WPARAMwIndex;

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

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

See Also LB_GETCARETINDEX

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:

```
HICONhIcon;

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:

```
HICONhIcon,hOldIcon;

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

See Also STM_GETICON

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_COMMNOTIFY 3.1

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

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_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 7, "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 7, "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 7, "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("salesperemployeeforeachdistrict")]
[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 7, "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_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_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_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_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_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_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_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 functionality by not passing the WM_WINDOWPOSCHANGING message to the **DefWindowProc** function.

See Also WM_WINDOWPOSCHANGED

Notification messages

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_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

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

C H A P T E R

7

Structures

ABC 3.1

The **ABC** structure contains the width of a character in a TrueType font.

```
typedef struct tagABC {   /* abc */
    int   abcA;
    UINT  abcB;
    int   abcC;
} ABC;
```

```
TABC = record
  abcA: Integer;
  abcB: Word;
  abcC: Integer;
end;
```

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**

CBT_CREATEWND 3.1

The **CBT_CREATEWND** structure contains information passed to a WH_CBT hook function before a window is created.

```
typedef struct tagCBT_CREATEWND {   /* cbtcw */
    CREATESTRUCT FAR* lpcs;
    HWND              hwndInsertAfter;
} CBT_CREATEWND;
```

```
TCBT_CreateWnd=record
  lpcs: PCreateStruct;
  hWndInsertAfter: HWnd;
end;
```

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 topmost window. If this parameter is 1, the window being created is the bottommost window.

See Also **CBTProc, SetWindowsHook**

CBTACTIVATESTRUCT 3.1

The **CBTACTIVATESTRUCT** structure contains information passed to a WH_CBT hook function before a window is activated.

```
typedef struct tagCBTACTIVATESTRUCT { /* cas */
    BOOL    fMouse;
    HWND    hWndActive;
} CBTACTIVATESTRUCT;
```

```
TCBTActivateStruct=record
  fMouse: Bool;
  hWndActive: HWnd;
end;
```

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

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.

```
#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;
```

```
TChooseColor = record
  lStructSize: Longint;
  hWndOwner: HWnd;
  hInstance: HWnd;
  rgbResult: Longint;
  lpCustColors: PLongint;
  Flags: Longint;
  lCustData: Longint;
  lpfnHook: function (Wnd: HWnd; Message, wParam: Word;
  lParam: Longint): Word;
  lpTemplateName: PChar;
end;
```

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.

Value	Meaning
CC_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.
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

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.

```
#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;
```

```
TChooseFont = record
  lStructSize: Longint;
  hWndOwner: HWnd;
  hDC: HDC;
  lpLogFont: PLogFont;
  iPointSize: Integer;
  Flags: Longint;
  rgbColors: Longint;
  lCustData: Longint;
  lpfnHook: function (Wnd: HWnd; Msg, wParam: Word; lParam: Longint):
    Word;
  lpTemplateName: PChar;
  hInstance: THandle;
  lpszStyle: PChar;
  nFontType: Word;
  nSizeMin: Integer;
  nSizeMax: Integer;
end;
```

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.
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.

Value	Meaning
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.
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.

Value	Meaning
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

The **CLASSENTRY** structure contains the name of a Windows class and a near pointer to the next class in the list.

```
#include <toolhelp.h>

typedef struct tagCLASSENTRY {  /* ce */
    DWORD   dwSize;
    HMODULE hInst;
    char    szClassName[MAX_CLASSNAME + 1];
    WORD    wNext;
} CLASSENTRY;
```

```
TClassEntry = record
  dwSize: Longint;
  hInst: THandle;
  szClassName: array[0..Max_ClassName] of Char;
  wNext: Word;
end;
```

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**

COMSTAT 3.1

The **COMSTAT** structure contains information about a communications device.

```
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;
```

```
TComStat = record
  Status: Byte;
  cbInQue: Word;             { count of characters in Rx Queue}
  cbOutQue: Word;            { count of characters in Tx Queue}
end;
```

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

The **CONVCONTEXT** structure contains information that makes it possible for applications to share data in several different languages.

```
#include <ddeml.h>

typedef struct tagCONVCONTEXT { /* cc                          */
    UINT       cb;
    UINT       wFlags;
    UINT       wCountryID;
    int        iCodePage;
    DWORD      dwLangID;
    DWORD      dwSecurity;
} CONVCONTEXT;
```

```
TConvContext = record
  cb: Word;
  wFlags: Word;
  wCountryID: Word;
  iCodePage: Integer;
  dwLangID: Longint;
  dwSecurity: Longint;
end;
```

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.
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

The **CONVINFO** structure contains information about a dynamic data exchange (DDE) conversation.

```
#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;
```

```
TConvInfo = record
  cb: Longint;
  hUser: Longint;
  hConvPartner: HConv;
  hszSvcPartner: HSZ;
  hszServiceReq: HSZ;
  hszTopic: HSZ;
  hszItem: HSZ;
  wFmt: Word;
  wType: Word;
  wStatus: Word;
  wConvst: Word;
  wLastError: Word;
  hConvList: HConvList;
  ConvCtxt: TConvContext;
end;
```

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.

Value	Meaning
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_BLOCKED
ST_BLOCKNEXT
ST_CLIENT
ST_CONNECTED
ST_INLIST
ST_ISLOCAL
ST_ISSELF
ST_TERMINATED

wConvst

Specifies the conversation state. This member can be one of the following values:

XST_ADVACKRCVD
XST_ADVDATAACKRCVD
XST_ADVDATASENT
XST_ADVSENT
XST_CONNECTED
XST_DATARCVD
XST_EXECACKRCVD
XST_EXECSENT
XST_INCOMPLETE
XST_INIT1
XST_INIT2
XST_NULL
XST_POKEACKRCVDX
ST_POKESENT
XST_REQSENT
XST_UNADVACKRCV
DXST_UNADVSENT

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

The **CPLINFO** structure contains resource information and a user-defined value for an extensible Control Panel application.

```
#include <cpl.h>

typedef struct tagCPLINFO { /* cpli */
    int   idIcon;
    int   idName;
    int   idInfo;
    LONG  lData;
} CPLINFO;
```

```
TCPLInfo = record
  idIcon: Integer;      { icon resource id, provided by CPlApplet() }
  idName: Integer;      { name string res. id, provided by CPlApplet() }
  idInfo: Integer;      { info string res. id, provided by CPlApplet() }
  lData: Longint;       { user defined data }
end;
```

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.

CTLINFO 3.1

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.

```
#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;
```

```
TCtlInfo = record
  wVersion: Word;        { control version }
  wCtlTypes: Word;       { control types }
  szClass: array[0..ctlClass-1] of Char;
                         { control class name }
  szTitle: array[0..ctlTitle-1] of Char;
                         { control title }
  szReserved: array[0..9] of Char;
                         { reserved for future use }
  ctType: array[0..ctlTypes-1] of TCtlType;
                         { control type list }
end;
```

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

The **CTLSTYLE** structure specifies the attributes of the selected control, including the current style flags, location, dimensions, and associated text.

```
#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;
```

```
TCtlStyle = record
  wX: Word;              { x origin of control }
  wY: Word;              { y origin of control }
  wCx: Word;             { width of control }
  wCy: Word;             { height of control }
  wId: Word;             { control child id }
  dwStyle: Longint;      { control style }
  szClass: array[0..ctlClass-1] of Char;
                         { name of control class }
  szTitle: array[0..ctlTitle-1] of Char;
                         { control text }
end;
```

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

The **CTLTYPE** structure contains information about a control in a particular class. The **CTLINFO** structure includes an array of **CTLTYPE** structures.

```
#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;
```

```
TCtlType = record
  wType: Word;          { type style }
  wWidth: Word;         { suggested width }
  wHeight: Word;        { suggested height }
  dwStyle: Longint;     { default style }
  szDescr: array[0..ctlDescr-1] of Char;
                        { menu name }
end;
```

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**

DDEACK 2.x

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.

```
#include <dde.h>

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

```
TDDEAck = record
  Flags: Word;
end;
```

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

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.

```
#include <dde.h>

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

```
TDDEAdvise = record
  Flags: Word;
  cfFormat: Integer;
end;
```

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

The **DDEDATA** structure contains the data and information about the data sent as part of a WM_DDE_DATA message.

```
#include <dde.h>

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

```
TDDEData=record
  Flags: Word;
  cfFormat: Integer;
  Value: array[0..0] of Char;
end;
```

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

The **DDEPOKE** structure contains the data and information about the data sent as part of a WM_DDE_POKE message.

```
#include <dde.h>

typedef struct tagDDEPOKE {   /* ddepok */
    WORD  unused:13,
          fRelease:1,
          fReserved:2;
    short cfFormat;
    BYTE  Value[1];
} DDEPOKE;
```

```
TDDEPoke = record
  Flags: Word;
  cfFormat: Word;
  Value: array[0..0] of Byte;
end;
```

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

The **DEBUGHOOKINFO** structure contains debugging information.

```
typedef struct tagDEBUGHOOKINFO {
    HMODULE hModuleHook;
    LPARAM  reserved;
    LPARAM  lParam;
    WPARAM  wParam;
    int     code;
} DEBUGHOOKINFO;
```

```
TDebugHookInfo = record
  hModuleHook: THandle;
  reserved: Longint;
  lParam: Longint;
  wParm: Word;
  code: Integer;
end;
```

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**

DEVNAMES 3.1

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.

```
#include <commdlg.h>

typedef struct tagDEVNAMES {    /* dn */
    UINT wDriverOffset;
    UINT wDeviceOffset;
    UINT wOutputOffset;
    UINT wDefault;
    /* optional data may appear here  */
} DEVNAMES;
```

```
TDevNames = record
  wDriverOffset: Word;
  wDeviceOffset: Word;
  wOutputOffset: Word;
  wDefault: Word;
end;
```

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

The **DOCINFO** structure contains the input and output filenames used by the **StartDoc** function.

```
typedef struct {    /* di */
    int     cbSize;
    LPCSTR  lpszDocName;
    LPCSTR  lpszOutput;
} DOCINFO;
```

```
TDocInfo = record
  cbSize: Integer;
  lpszDocName: PChar;
  lpszOutput: PChar;
end;
```

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**

DRIVERINFOSTRUCT 3.1

The **DRIVERINFOSTRUCT** structure contains basic information about an installable device driver.

```
typedef struct tagDRIVERINFOSTRUCT {     /* drvinfst */
    UINT      length;
    HDRVR     hDriver;
    HINSTANCE hModule;
    char      szAliasName[128];
} DRIVERINFOSTRUCT;
```

```
TDriverInfoStruct=record
  length: Word;
  hDriver: THandle;
  hModule: THandle;
  szAliasName: array[0..128] of Char;
end;
```

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

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.

```
typedef struct tagDRVCONFIGINFO {
    DWORD   dwDCISize;
    LPCSTR  lpszDCISectionName;
    LPCSTR  lpszDCIAliasName;
} DRVCONFIGINFO;
```

```
TDrvConfigInfo=record
  dwDCISize: Longint;
  lpszDCISectionName: PChar;
  lpszDCIAliasName: PChar;
end;
```

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

The **EVENTMSG** structure contains information from the Windows application queue. This structure is used to store message information for the **JournalPlaybackProc** callback function.

```
typedef struct tagEVENTMSG {    /* em */
    UINT  message;
    UINT  paramL;
    UINT  paramH;
    DWORD time;
} EVENTMSG;
```

```
TEventMsg = record
  message: Word;
  paramL: Word;
  paramH: Word;
  time: Longint;
end;
```

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

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.

```
#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;
```

```
TFindReplace = record
  lStructSize: Longint;
  hWndOwner: HWnd;
  hInstance: THandle;
  Flags: Longint;
  lpstrFindWhat: PChar;
  lpstrReplaceWith: PChar;
  wFindWhatLen: Word;
  wReplaceWithLen: Word;
  lCustData: Longint;
  lpfnHook: function (Wnd: HWnd; Msg, wParam: Word; lParam: Longint):
                    Word;
  lpTemplateName: PChar;
end;
```

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 pre-loaded 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

The **FIXED** structure contains the integral and fractional parts of a fixed-point real number.

```
typedef struct tagFIXED {   /* fx */
    UINT    fract;
    int     value;
} FIXED;
```

```
TFixed = record
  fract: Word;
  value: Integer;
end;
```

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

The **FMS_GETDRIVEINFO** structure contains information about the drive that is selected in the currently active File Manager window.

```
#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;
```

```
TGetDriveInfo = record
  dwTotalSpace: Longint;
  dwFreeSpace: Longint;
  szPath: array[0..259] of Char; { current directory }
  szVolume: array[0..13] of Char; { volume label }
  szShare: array[0..127] of Char; { if this is a net drive }
end;
```

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

The **FMS_GETFILESEL** structure contains information about a selected file in File Manager's directory window or Search Results window.

```
#include <wfext.h>

typedef struct tagFMS_GETFILESEL { /* fmsgfs */
    UINT  wTime;
    UINT  wDate;
    DWORD dwSize;
    BYTE  bAttr;
    char  szName[260];
} FMS_GETFILESEL;
```

```
TGetFileSel = record
  wTime: Word;
  wDate: Word;
  dwSize: Longint;
  bAttr: Byte;
  szName: array[0..259] of Char;        { always fully qualified }
end;
```

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

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.

```
#include <wfext.h>

typedef struct tagFMS_LOAD { /* fmsld */
    DWORD dwSize;
    char  szMenuName[MENU_TEXT_LEN];
    HMENU hMenu;
    UINT  wMenuDelta;
} FMS_LOAD;
```

```
TFMS_Load = record
  dwSize: Longint;      { for version checks }
  szMenuName: array[0..Menu_Text_Len-1] of Char;  { output }
  Menu: HMenu;          { output }
  wMenuDelta: Word;         { input }
end;
```

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* parameter 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

The **GLOBALENTRY** structure contains information about a memory object on the global heap.

```
#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;
```

```
TGlobalEntry = record
  dwSize: Longint;
  dwAddress: Longint;
  dwBlockSize: Longint;
  hBlock: THandle;
  wcLock: Word;
  wcPageLock: Word;
  wFlags: Word;
  wHeapPresent: Bool;
  hOwner: THandle;
  wType: Word;
  wData: Word;
  dwNext: Longint;
  dwNextAlt: Word;
end;
```

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.

Value	Meaning
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.
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.

Value	Meaning
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

The **GLOBALINFO** structure contains information about the global heap.

```
#include <toolhelp.h>

typedef struct tagGLOBALINFO {  /* gi */
    DWORD dwSize;
    WORD  wcItems;
    WORD  wcItemsFree;
    WORD  wcItemsLRU;
} GLOBALINFO;
```

```
TGlobalInfo = record
  dwSize: Longint;
  wcItems: Word;
  wcItemsFree: Word;
  wcItemsLRU: Word;
end;
```

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

The **GLYPHMETRICS** structure contains information about the placement and orientation of a glyph in a character cell.

```
typedef struct tagGLYPHMETRICS {  /* gm */
    UINT  gmBlackBoxX;
    UINT  gmBlackBoxY;
    POINT gmptGlyphOrigin;
    int   gmCellIncX;
    int   gmCellIncY;
} GLYPHMETRICS;
```

```
TGlyphMetrics = record
  gmBlackBoxX: Word;
  gmBlackBoxY: Word;
  gmptGlyphOrigin: TPoint;
  gmCellIncX: Integer;
  gmCellIncY: Integer;
end;
```

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**

HARDWAREHOOKSTRUCT 3.1

The **HARDWAREHOOKSTRUCT** contains information about a hardware message placed in the system message queue.

```
typedef struct tagHARDWAREHOOKSTRUCT {  /* hhs */
    HWND    hWnd;
    UINT    wMessage;
    WPARAM  wParam;
    LPARAM  lParam;
} HARDWAREHOOKSTRUCT;
```

```
THardwareHookStruct=record
  hWnd: HWnd;
  wMessage: Word;
  wParam: Word;
  lParam: Longint;
end;
```

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

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.

```
typedef struct {
    int  wStructSize;
    int  x;
    int  y;
    int  dx;
    int  dy;
    int  wMax;
    char rgchMember[2];
} HELPWININFO;
```

```
THelpWinInfo = record
  wStructSize: Integer;
  x: Integer;
  y: Integer;
  dx: Integer;
  dy: Integer;
  wMax: Integer;
  rgchMember: array[0..1] of Char;
end;
```

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

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.

```
#include <ddeml.h>

typedef struct tagHSZPAIR {   /* hp */
    HSZ hszSvc;
    HSZ hszTopic;
} HSZPAIR;
```

```
THSZPair = record
  hszSvc: HSZ;
  hszTopic: HSZ;
end;
```

Members

hszSvc	Identifies a service name.
hszTopic	Identifies a topic name.

KERNINGPAIR 3.1

The **KERNINGPAIR** structure defines a kerning pair.

```
typedef struct tagKERNINGPAIR {
    WORD wFirst;
    WORD wSecond;
    int  iKernAmount;
} KERNINGPAIR;
```

```
TKerningPair = record
  wFirst: Word;
  wSecond: Word;
  iKernAmount: Integer;
end;
```

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

The **LOCALENTRY** structure contains information about a memory object on the local heap.

```
#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;
```

```
TLocalEntry = record
  dwSize: Longint;
  hHandle: THandle;
  wAddress: Word;
  wSize: Word;
  wFlags: Word;
  wcLock: Word;
  wType: Word;
  hHeap: Word;
  wHeapType: Word;
  wNext: Word;
end;
```

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.

Value	Meaning
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.
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

The **LOCALINFO** structure contains information about the local heap.

```
#include <toolhelp.h>

typedef struct tagLOCALINFO {  /* li */
    DWORD  dwSize;
    WORD   wcItems;
} LOCALINFO;
```

```
TLocalInfo = record
  dwSize: Longint;
  wcItems: Word;
end;
```

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**

MAT2 3.1

The **MAT2** structure contains the values for a transformation matrix.

```
typedef struct tagMAT2 {  /* mat2 */
    FIXED eM11;
    FIXED eM12;
    FIXED eM21;
    FIXED eM22;
} MAT2;
```

```
TMat2 = record
  eM11: TFixed;
  eM12: TFixed;
  eM21: TFixed;
  eM22: TFixed;
end;
```

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**

MEMMANINFO 3.1

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.

```
#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;
```

```
TMemManInfo = record
  dwSize: Longint;
  dwLargestFreeBlock: Longint;
  dwMaxPagesAvailable: Longint;
  dwMaxPagesLockable: Longint;
  dwTotalLinearSpace: Longint;
  dwTotalUnlockedPages: Longint;
  dwFreePages: Longint;
  dwTotalPages: Longint;
  dwFreeLinearSpace: Longint;
  dwSwapFilePages: Longint;
  wPageSize: Word;
end;
```

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**

METAHEADER 3.1

The **METAHEADER** structure contains information about a metafile.

```
typedef struct tagMETAHEADER {  /* mh */
    UINT  mtType;
    UINT  mtHeaderSize;
    UINT  mtVersion;
    DWORD mtSize;
    UINT  mtNoObjects;
    DWORD mtMaxRecord;
    UINT  mtNoParameters;
} METAHEADER;
```

```
TMetaHeader= record
  mtType : Word;
  mtHeaderSize : Word;
  mtVersion : Word;
  mtSize : Longint;
  mtNoObjects : Word;
  mtMaxRecord : Longint;
  mtNoParameters : Word;
end;
```

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

The **METARECORD** structure contains a metafile record.

```
typedef struct tagMETARECORD {  /* mr */
    DWORD rdSize;
    UINT  rdFunction;
    UINT  rdParm[1];
} METARECORD;
```

```
TMetaRecord = record
  rdSize: Longint;
  rdFunction: Word;
  rdParm: array[0..0] of Word;
end;
```

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

The **MINMAXINFO** structure contains information about a window's maximized size and position and its minimum and maximum tracking size.

```
typedef struct tagMINMAXINFO {  /* mmi */
    POINT ptReserved;
    POINT ptMaxSize;
    POINT ptMaxPosition;
    POINT ptMinTrackSize;
    POINT ptMaxTrackSize;
} MINMAXINFO;
```

```
TMinMaxInfo = record
  ptReserved: TPoint;
  ptMaxSize: TPoint;
  ptMaxPosition: TPoint;
  ptMinTrackSize: TPoint;
  ptMaxTrackSize: TPoint;
end;
```

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

The **MODULEENTRY** structure contains information about one module in the module list.

```
#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;
```

```
TModuleEntry=record
  dwSize: Longint;
  szModule : array[0..max_Module_Name] of Char;
  hModule: THandle;
  wUsageFlags: Word;
  szExePath: array[0..max_Path] of Char;
  wNext: Word;
end;
```

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

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.

```
#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;
```

```
TMonCBStruct = record
  cb: Word;
  wReserved: Word;
  dwTime: Longint;
  hTask: THandle;
  dwRet: Longint;
  wType: Word;
  wFmt: Word;
  hConv: HConv;
  hsz1: HSZ;
  hsz2: HSZ;
  hData: HDDEData;
  dwData1: Longint;
  dwData2: Longint;
end;
```

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

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.

```
#include <ddeml.h>

typedef struct tagMONCONVSTRUCT { /* mcvst */
    UINT    cb;
    BOOL    fConnect;
    DWORD   dwTime;
    HANDLE  hTask;
    HSZ     hszSvc;
    HSZ     hszTopic;
    HCONV   hConvClient;
    HCONV   hConvServer;
} MONCONVSTRUCT;
```

```
TMonConvStruct = record
  cb: Word;
  fConnect: Bool;
  dwTime: Longint;
  hTask: THandle;
  hszSvc: HSz;
  hszTopic: HSz;
  hConvClient: HConv;
  hConvServer: HConv;
end;
```

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

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.

```
#include <ddeml.h>

typedef struct tagMONERRSTRUCT {  /* mest */
    UINT    cb;
    UINT    wLastError;
    DWORD   dwTime;
    HANDLE  hTask;
} MONERRSTRUCT;
```

```
TMonErrStruct = record
  cb: Word;
  wLastError: Word;
  dwTime: Longint;
  hTask: THandle;
end;
```

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

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).

```
#include <ddeml.h>

typedef struct tagMONHSZSTRUCT { /* mhst */
    UINT   cb;
    BOOL   fsAction;
    DWORD  dwTime;
    HSZ    hsz;
    HANDLE hTask;
    WORD   wReserved;
    char   str[1];
} MONHSZSTRUCT;
```

```
TMonHSZStruct = record
  cb: Word;
  fsAction: Bool;      { mh_ value }
  dwTime: Longint;
  HSZ: HSZ;
  hTask: THandle;
  wReserved: Word;
  Str: array[0..0] of Char;
end;
```

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

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.

```
#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;
```

```
TMonLinkStruct = record
  cb: Word;
  dwTime: Longint;
  hTask: THandle;
  fEstablished: Bool;
  fNoData: Bool;
  hszSvc: HSz
  hszTopic: HSz;
  hszItem: HSz;
  wFmt: Word;
  fServer: Bool;
  hConvServer: HConv;
  hConvClient: HConv;
end;
```

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

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.

```
#include <ddeml.h>

typedef struct tagMONMSGSTRUCT { /* mmst */
    UINT    cb;
    HWND    hwndTo;
    DWORD   dwTime;
    HANDLE  hTask;
    UINT    wMsg;
    WPARAM  wParam;
    LPARAM  lParam;
} MONMSGSTRUCT;
```

```
TMonMsgStruct = record
  cb: Word;
  hWndTo: HWnd;
  dwTime: Longint;
  hTask: THandle;
  wMsg: Word;
  wParam: Word;
  lParam: Longint;
end;
```

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

The **MOUSEHOOKSTRUCT** structure contains information about a mouse event.

```
typedef struct tagMOUSEHOOKSTRUCT { /* ms */
    POINT   pt;
    HWND    hwnd;
    UINT    wHitTestCode;
    DWORD   dwExtraInfo;
} MOUSEHOOKSTRUCT;
```

```
TMouseHookStruct = record
  pt: TPoint;
  hWnd: HWnd;
  wHitTestCode: Word;
  dwExtraInfo: Longint;
end;
```

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 retrieve this information by calling the **GetMessageExtraInfo** function.

See Also **GetMessageExtraInfo, SetWindowsHook**

NCCALCSIZE_PARAMS 3.1

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.

```
typedef struct tagNCCALCSIZE_PARAMS {
    RECT          rgrc[3];
    WINDOWPOS FAR* lppos;
} NCCALCSIZE_PARAMS;
```

```
TNCCalcSize_Params=record
  rgrc: array[0..2] of TRect;
  lppos: PWindowPos;
end;
```

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

The **NEWCPLINFO** structure contains resource information and a user-defined value for a Control Panel application.

```
#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;
```

```
TNewCPLInfo= record
  dwSize: Longint;                      { similar to the commdlg }
  dwFlags: Longint;
  dwHelpContext: Longint;               { help context to use }
  lData: Longint;                       { user defined data }
  Icon: HIcon; { icon to use, this is owned by CONTROL.EXE (may be
                    deleted) }
  szName: array[0..31] of Char;         { short name }
  szInfo: array[0..63] of Char;         { long name (status line) }
  szHelpFile: array[0..127] of Char;    { path to help file to use }
end;
```

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.
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

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.

```
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;
```

```
TNewTextMetric = record
  tmHeight: Integer;
  tmAscent: Integer;
  tmDescent: Integer;
  tmInternalLeading: Integer;
  tmExternalLeading: Integer;
  tmAveCharWidth: Integer;
  tmMaxCharWidth: Integer;
  tmWeight: Integer;
  tmItalic: Byte;
  tmUnderlined: Byte;
  tmStruckOut: Byte;
  tmFirstChar: Byte;
  tmLastChar: Byte;
  tmDefaultChar: Byte;
  tmBreakChar: Byte;
  tmPitchAndFamily: Byte;
  tmCharSet: Byte;
  tmOverhang: Integer;
  tmDigitizedAspectX: Integer;
  tmDigitizedAspectY: Integer;
  ntmFlags: Longint;            { various flags (fsSelection) }
  ntmSizeEM: Word;              { size of EM }
  ntmCellHeight: Word;          { height of font in notional units }
  ntmAvgWidth: Word;            { average with in notional units }
end;
```

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 width of the widest character in the font.

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, Courier New, the TMPF_PITCH, TMPF_VECTOR, and TMPF_TT bits would be set for the monospace TrueType font.

When the TMPF_TT 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_TT 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.
FF_SCRIPT	Fonts designed to look like handwriting. Script and Cursive are examples.

Value	Meaning
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

The **NFYLOADSEG** structure contains information about the segment being loaded when the kernel sends a load-segment notification.

```
#include <toolhelp.h>

typedef struct tagNFYLOADSEG {  /* nfyls */
    DWORD   dwSize;
    WORD    wSelector;
    WORD    wSegNum;
    WORD    wType;
    WORD    wcInstance;
    LPCSTR  lpstrModuleName;
} NFYLOADSEG;
```

```
TNFYLoadSeg = record
  dwSize: Longint;
  wSelector: Word;
  wSegNum: Word;
  wType: Word;
  hInstance: THandle;
  lpstrModuleName: PChar;
end;
```

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 Identifies the application instance being loaded.

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

The **NFYLOGERROR** structure contains information about a validation error that caused the kernel to send an NFY_LOGERROR notification.

```
#include <toolhelp.h>

typedef struct tagNFYLOGERROR {  /* nfyle */
    DWORD      dwSize;
    UINT       wErrCode;
    void FAR* lpInfo;
} NFYLOGERROR;
```

```
TNFYLogError = record
  dwSize: Longint;
  wErrCode: Word;
  lpInfo: PChar;       { Error code-dependent }
end;
```

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

The **NFYLOGPARAMERROR** structure contains information about a parameter-validation error that caused the kernel to send an NFY_LOGPARAMERROR notification.

```
#include <toolhelp.h>

typedef struct tagNFYLOGPARAMERROR {   /* nfylpe */
    DWORD          dwSize;
    UINT           wErrCode;
    FARPROC        lpfnErrorAddr;
    void FAR* FAR* lpBadParam;
} NFYLOGPARAMERROR;
```

```
TNFYLogParamError=record
  dwSize: Longint;
  wErrCode: Word;
  lpfnErrorAddr : TFarProc;
  lpBadParam: PChar;
end;
```

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

The **NFYRIP** structure contains information about the system when a system debugging error (RIP) occurs.

```
#include <toolhelp.h>

typedef struct tagNFYRIP {   /* nfyr */
    DWORD dwSize;
    WORD wIP;
    WORD wCS;
    WORD wSS;
    WORD wBP;
    WORD wExitCode;
} NFYRIP;
```

```
TNFYRip = record
  dwSize: Longint;
  wIP: Word;
  wCS: Word;
  wSS: Word;
  wBP: Word;
wExitCode:Word;
end;
```

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

The **NFYSTARTDLL** structure contains information about the dynamic-link library (DLL) being loaded when the kernel sends a load-DLL notification.

```
#include <toolhelp.h>

typedef struct tagNFYSTARTDLL {  /* nfysd */
    DWORD    dwSize;
    HMODULE hModule;
    WORD     wCS;
    WORD     wIP;
} NFYSTARTDLL;
```

```
TNFYStartDLL = record
  dwSize: Longint;
  hModule: THandle;
  wCS: Word;
  wIP: Word;
end;
```

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**

OLECLIENT 3.1

The **OLECLIENT** structure points to an **OLECLIENTVTBL** structure and can store state information for use by the client application.

```
#include <ole.h>

typedef struct _OLECLIENT { /* oc */
    LPOLECLIENTVTBL lpvtbl;
    .
    .  /* any client-supplied state information */
    .
} OLECLIENT;
```

```
TOleClient = record
  lpvtbl: POleClientVTBL;
end;
```

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

The **OLECLIENTVTBL** structure contains a pointer to a callback function for the client application.

```
#include <ole.h>

typedef struct _OLECLIENTVTBL {     /* ocv */
    int (CALLBACK* CallBack)(LPOLECLIENT, OLE_NOTIFICATION,
LPOLEOBJECT);
} OLECLIENTVTBL;
```

```
TOleClientVTbl = record
  CallBack: function (Client: POleClient; Nofication: TOle_Notification;
  OleObject: POleObject): Integer;
end;
```

Comments The address passed as the **CallBack** member must be created by using the **MakeProcInstance** function.

Function
ClientCallback

Syntax INT ClientCallback(lpclient, notification, 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.

Value	Meaning
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..
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

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.

```
#include <ole.h>

typedef struct _OLEOBJECT {      /* oo */
    LPOLEOBJECTVTBL lpvtbl;
    .
    .  /* any server-supplied state information */
    .
} OLEOBJECT;
```

```
TOleObject = record
  lpvtbl: POleObjectVTbl;
end;
```

Members

lpvtbl	Points to a table of function pointers for the object.

OLEOBJECTVTBL 3.1

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.

```
#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;
```

```
TOleObjectVTbl = record
    QueryProtocol: function (Self: POleObject; Protocol: PChar):
       Pointer;
    Release: function (Self: POleObject): TOleStatus;
    Show: function (Self: POleObject; TakeFocus: Bool): TOleStatus;
    DoVerb: function (Self: POleObject; Verb: Word; Show, Focus: Bool):
       TOleStatus;
    GetData: function (Self: POleObject; Format: TOleClipFormat;
      var Handle: THandle): TOleStatus;
    SetData: function (Self: POleObject; Format: TOleClipFormat;
      Data: THandle): TOleStatus;
    SetTargetDevice: function (Self: POleObject;
      TargetDevice: THandle): TOleStatus;
    SetBounds: function (Self: POleObject; var Bounds: TRect):
      TOleStatus;
    EnumFormats: function (Self: POleObject;
      Format: TOleClipFormat): TOleClipFormat;
    SetColorScheme: function (Self: POleObject; var Palette:
      TLogPalette): TOleStatus;

    { Server has to implement only the above methods. }

    { Extra methods required for client. }

    Delete: function (Self: POleObject): TOleStatus;
    SetHostNames: function (Self: POleObject; Client,
      ClientObj: PChar): TOleStatus;
    SaveToStream: function (Self: POleObject; Stream: POleStream):
      TOleStatus;
    Clone: function (Self: POleObject; Client: POleClient;
      ClientDoc: LHClientDoc; ObjectName: PChar;
      var OleObject: POleObject): TOleStatus;
    CopyFromLink: function (Self: POleObject; Client: POleClient;
      ClientDoc: LHClientDoc; ObjName: PChar;
      var OleObject: POleObject): TOleStatus;
    Equal: function (Self: POleObject; OleObject: POleObject):
      TOleStatus;
    CopyToClipboard: function (Self: POleObject): TOleStatus;
    Draw: function (Self: POleObject; DC: HDC; var Bounds, WBounds:
      TRect; FormatDC: HDC): TOleStatus;
    Activate: function (Self: POleObject; Verb: Word; Show, TakeFocus:
      Bool; hWnd: HWnd; Bounds: PRect): TOleStatus;
    Execute: function (Self: POleObject; Commands: THandle;
      Reserved: Word): TOleStatus;
    Close: function (Self: POleObject): TOleStatus;
    Update: function (Self: POleObject): TOleStatus;
    Reconnect: function (Self: POleObject): TOleStatus;
```

```
    ObjectConvert: function (Self: POleObject; Protocol: PChar;
      Client: POleClient; ClientDoc: LHClientDoc; ObjName: PChar;
      var OleObject: POleObject): TOleStatus;
    GetLinkUpdateOptions: function (Self: POleObject;
      var UpdateOpt: TOleOpt_Update): TOleStatus;
    SetLinkUpdateOptions: function (Self: POleObject;
      UpdateOpt: TOleOpt_Update): TOleStatus;

    Rename: function (Self: POleObject; NewName: PChar): TOleStatus;
    QueryName: function (Self: POleObject; Name: PChar;
      var NameSize: Word): TOleStatus;

    QueryType: function (Self: POleObject; var ObjType: Longint):
      TOleStatus;
    QueryBounds: function (Self: POleObject; var Bounds: TRect):
      TOleStatus;
    QuerySize: function (Self: POleObject; var Size: Longint):
      TOleStatus;
    QueryOpen: function (Self: POleObject): TOleStatus;
    QueryOutOfDate: function (Self: POleObject): TOleStatus;

    QueryReleaseStatus: function (Self: POleObject): TOleStatus;
    QueryReleaseError: function (Self: POleObject): TOleStatus;
    QueryReleaseMethod: function (Self: POleObject):
      TOle_Release_Method;

    RequestData: function (Self: POleObject;
      Format: TOleClipFormat): TOleStatus;
    ObjectLong: function (Self: POleObject; Flags: Word;
      Data: PLongint): TOleStatus;

    { This method is internal only }
    ChangeData: function (Self: POleObject; Data: THandle;
     Client: POleClient; Flag: Bool): TOleStatus;
  end;
```

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

Syntax OLESTATUS (FAR PASCAL *Release)(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

Syntax OLESTATUS (FAR PASCAL *Show)(lpObject, fTakeFocus)
function Show(Self: POleObject; TakeFocus: Bool): TOleStatus;

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

Syntax OLESTATUS (FAR PASCAL *DoVerb)(lpObject, iVerb, fShow, 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

Syntax OLESTATUS (FAR PASCAL *GetData)(lpObject, cfFormat, 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

Syntax OLESTATUS (FAR PASCAL *SetData)(lpObject, cfFormat, 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

Syntax

OLESTATUS (FAR PASCAL *SetTargetDevice)(lpObject, 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

Syntax

OLESTATUS (FAR PASCAL *ObjectLong)(lpObject, wFlags, 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

Syntax OLESTATUS SetColorScheme(lpObject, 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

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.

```
#include <ole.h>

typedef struct _OLESERVER {     /* os */
    LPOLESERVERVTBL lpvtbl;
    .
    .  /* any server-supplied state information */
    .
} OLESERVER;
```

```
TOleServer = record
  lpvtbl: POleServerVTbl;
end;
```

Members	**lpvtbl**	Points to a table of function pointers for the server.

OLESERVERDOC 3.1

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.

```
#include <ole.h>

typedef struct _OLESERVERDOC {  /* osd */
    LPOLESERVERDOCVTBL lpvtbl;
    .
    .  /* any server-supplied document-state information */
    .

} OLESERVERDOC;
```

```
TOleServerDoc = record
  lpvtbl: POleServerDocVTbl;
end;
```

Members **lpvtbl** Points to a table of function pointers for the document.

OLESERVERDOCVTBL 3.1

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.

```
#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;
```

```
TOleServerDocVTbl=record
  Save: function (Doc: POleServerDoc): TOleStatus;
  Close: function (Doc: POleServerDoc): TOleStatus;
  SetHostNames: function (Doc: POleServerDoc; Client, Doc: PChar):
     TOleStatus;
  SetDocDimensions: function (Doc: POleServerDoc;
     var Bounds: TRect): TOleStatus;
  GetObject: function (Doc: POleServerDoc; Item: PChar;
     var OleObject: POleObject; Client: POleClient): TOleStatus;
  Release: function (Doc: POleServerDoc): TOleStatus;
  SetColorScheme: function (Doc: POleServerDoc;
     var Palette: TLogPalette): TOleStatus;
  Execute: function (Doc: POleServerDoc; Commands: THandle):
     TOleStatus;
end;
```

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

Syntax OLESTATUS Save(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

Syntax OLESTATUS Close(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

Syntax OLESTATUS SetHostNames(lpDoc, lpszClient, 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**

Syntax OLESTATUS SetDocDimensions(lpDoc, 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**

Syntax OLESTATUS GetObject(lpDoc, lpszItem, lplpObject, 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

Syntax OLESTATUS Release(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

Syntax OLESTATUS SetColorScheme(lpDoc, 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**

Syntax OLESTATUS Execute(lpDoc, 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

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.

```
#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;
```

```
TOleServerVTbl = record
  Open: function (Server: POleServer; Doc: LHServerDoc; DocName: PChar;
    var ServerDoc: POleServerDoc): TOleStatus;
  Create: function (Server: POleServer; Doc: LHServerDoc; Class,
    DocName: PChar; var ServerDoc: POleServerDoc): TOleStatus;
  CreateFromTemplate: function (Server: POleServer; Doc: LHServerDoc;
    Class, DocName, TemplateName: PChar; var ServerDoc: POleServerDoc):
    TOleStatus;
  Edit: function (Server: POleServer; Doc: LHServerDoc; Class,
    DocName: PChar; var ServerDoc: POleServerDoc): TOleStatus;
  Exit: function (Server: POleServer): TOleStatus;
  Release: function (Server: POleServer): TOleStatus;
  Execute: function (Server: POleServer; Commands: THandle): TOleStatus;
end;
```

Every function except **Release** can return OLE_BUSY.

Function Open

Syntax OLESTATUS Open(lpServer, lhDoc, lpszDoc, 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

Syntax OLESTATUS Create(lpServer, lhDoc, lpszClass, lpszDoc, 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

Syntax OLESTATUS CreateFromTemplate(lpServer, lhDoc, lpszClass, lpszDoc, lpszTemplate, 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**

Syntax OLESTATUS Edit(lpServer, lhDoc, lpszClass, lpszDoc, 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

Syntax OLESTATUS Exit(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

Syntax OLESTATUS Release(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

Syntax OLESTATUS Execute(lpServer, 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

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.

```
#include <ole.h>

typedef struct _OLESTREAM {      /* ostr */
    LPOLESTREAMVTBL lpstbl;
} OLESTREAM;
```

```
TOleStream = record
  lpstbl: POleStreamVTbl;
end;
```

Members **lpstbl** Points to an **OLESTREAMVTBL** structure.

OLESTREAMVTBL 3.1

The **OLESTREAMVTBL** structure points to functions the client library uses for stream operations on objects. This structure is allocated and initialized by client applications.

```
#include <ole.h>

typedef struct _OLESTREAMVTBL {   /* ostrv */
    DWORD (CALLBACK* Get)(LPOLESTREAM, void FAR*, DWORD);
    DWORD (CALLBACK* Put)(LPOLESTREAM, OLE_CONST void FAR*, DWORD);
} OLESTREAMVTBL;
```

```
TOleStreamVTbl = record
  Get: function (Stream: POleStream; Buffer: PChar; Size: Longint):
     Longint;
  Put: function (Stream: POleStream; Buffer: PChar; Size: Longint):
     Longint;
end;
```

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**

Syntax DWORD Get(lpstream, lpszBuf, 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**

Syntax DWORD Put(lpstream, lpszBuf, 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

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.

```
#include <ole.h>

typedef struct _OLETARGETDEVICE {
    UINT otdDeviceNameOffset;
    UINT otdDriverNameOffset;
    UINT otdPortNameOffset;
    UINT otdExtDevmodeOffset;
    UINT otdExtDevmodeSize;
    UINT otdEnvironmentOffset;
    UINT otdEnvironmentSize;
    BYTE otdData[1];
} OLETARGETDEVICE;
```

```
TOleTargetDevice=record
  otdDeviceNameOffset: Word;
  otdDriverNameOffset: Word;
  otdPortNameOffset: Word;
  otdExtDevmodeOffset: Word;
  otdExtDevmodeSize: Word;
  otdEnvironmentOffset: Word;
  otdEnvironmentSize: Word;
  otdData: array[0..0] of Byte;
end;
```

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

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.

```
#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;
```

```
TOpenFilename = record
  lStructSize: Longint;
  hWndOwner: HWnd;
  hInstance: THandle;
  lpstrFilter: PChar;
  lpstrCustomFilter: PChar;
  nMaxCustFilter: Longint;
  nFilterIndex: Longint;
  lpstrFile: PChar;
  nMaxFile: Longint;
  lpstrFileTitle: PChar;
  nMaxFileTitle: Longint;
  lpstrInitialDir: PChar;
  lpstrTitle: PChar;
  Flags: Longint;
  nFileOffset: Word;
  nFileExtension: Word;
  lpstrDefExt: PChar;
  lCustData: Longint;
  lpfnHook: function (Wnd: HWnd; Msg, wParam: Word; lParam: Longint):
    Word;
  lpTemplateName: PChar;
end;
```

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

Value	Meaning
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.
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.

Value	Meaning
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.

Value	Meaning
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.
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.

Value	Meaning
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. 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

The **OUTLINETEXTMETRIC** structure contains metrics describing a TrueType font.

```
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;
```

```
TOutlineTextMetric=record
  otmSize: Word;                { I size of this structure                 }
  otmTextMetrics: TTextMetric;  { regular text metrics                     }
  otmFiller: Byte;              { want to be word aligned                  }
  otmPanoseNumber: TPanose;     { Panose number of font                    }
  otmfsSelection: Word;         { B Font selection flags (see #defines)    }
  otmfsType: Word;              { B Type indicators   (see #defines)       }
  otmsCharSlopeRise: Word;      { Slope angle Rise / Run   1 vertical      }
  otmsCharSlopeRun: Word;       {                  0 vertical              }
  otmEMSquare: Word;            { N size of EM                             }
  otmAscent: Word;              { D ascent above baseline                  }
  otmDescent: Word;             { D descent below baseline                 }
  otmLineGap: Word;             { D                                        }
  otmCapEmHeight: Word;         { D height of upper case M                 }
  otmXHeight: Word;             { D height of lower case chars in font     }
  otmrcFontBox: TRect;          { D Font bounding box                      }
  otmMacAscent: Word;           { D ascent above baseline for Mac          }
  otmMacDescent: Word;          { D descent below baseline for Mac         }
  otmMacLineGap: Word;          { D                                        }
  otmusMinimumPPEM: Word;       { D Minimum point ppem                     }
  otmptSubscriptSize: TPoint;   { D Size of subscript                      }
  otmptSubscriptOffset: TPoint; { D Offset of subscript                    }
  otmptSuperscriptSize: TPoint; { D Size of superscript                    }
  otmptSuperscriptOffset: TPoint;{ D Offset of superscript                 }
  otmsStrikeoutSize: Word;      { D Strikeout size                         }
  otmsStrikeoutPosition: Word;  { D Strikeout position                     }
  otmsUnderscoreSize: Word;     { D Underscore size                        }
  otmsUnderscorePosition: Word; { D Underscore position                    }
  otmpFamilyName: PChar;        { offset to family name                    }
  otmpFaceName: PChar;          { offset to face name                      }
  otmpStyleName: PChar;         { offset to Style string                   }
  otmpFullName: PChar;          { offset to full name                      }
end;
```

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**

PANOSE 3.1

The **PANOSE** structure describes the Panose font-classification values for a TrueType font.

```
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;
```

```
TPanose = record
  bFamilyType: Byte;
  bSerifStyle: Byte;
  bWeight: Byte;
  bProportion: Byte;
  bContrast: Byte;
  bStrokeVariation: Byte;
  bArmStyle: Byte;
  bLetterform: Byte;
  bMidline: Byte;
  bXHeight: Byte;
end;
```

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

Value	Meaning
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

Value	Meaning
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

POINTFX 3.1

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.

```
typedef struct tagPOINTFX {
    FIXED x;
    FIXED y;
} POINTFX;
```

```
TPointFX = record
  x: TFixed;
  y: TFixed;
end;
```

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

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.

```
#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;
```

```
TPrintDlg = record
  lStructSize: Longint;
  hWndOwner: HWnd;
  hDevMode: THandle;
  hDevNames: THandle;
  hDC: HDC;
  Flags: Longint;
  nFromPage: Word;
  nToPage: Word;
  nMinPage: Word;
  nMaxPage: Word;
  nCopies: Word;
  hInstance: THandle;
  lCustData: Longint;
  lpfnPrintHook: function (Wnd: HWnd; Msg, wParam: Word;
    lParam: Longint): Integer;
  lpfnSetupHook: function (Wnd: HWnd; Msg, wParam: Word;
    lParam: Longint): Integer;
  lpPrintTemplateName: PChar;
  lpSetupTemplateName: PChar;
  hPrintTemplate: THandle;
  hSetupTemplate: THandle;
end;
```

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/HPLaserJet=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 PC_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.

Value	Meaning
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.
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.

Value	Meaning
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.
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.

Value	Meaning
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_PAGE NUMS 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 pre-loaded 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 pre-loaded 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_ENABLEPRINTTEMPLATE-HANDLE flag must be set. This member is filled on input.

See Also **CreateDC, CreateIC, PrintDlg, DEVMODE, DEVNAMES**

RASTERIZER_STATUS 3.1

The **RASTERIZER_STATUS** structure contains information about whether TrueType is installed. This structure is filled when an application calls the **GetRasterizerCaps** function.

```
typedef struct tagRASTERIZER_STATUS {    /* rs */
    int   nSize;
    int   wFlags;
    int   nLanguageID;
} RASTERIZER_STATUS;
```

```
TRasterizer_Status=record
  nSize: Integer;
  wFlags: Integer;
  nLanguageID: Integer;
end;
```

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**

SEGINFO 3.1

The **SEGINFO** structure contains information about a data or code segment. This structure is filled in by the **GetCodeInfo** function.

```
typedef struct tagSEGINFO {
    UINT    offSegment;
    UINT    cbSegment;
    UINT    flags;
    UINT    cbAlloc;
    HGLOBAL h;
    UINT    alignShift;
    UINT    reserved[2];
} SEGINFO;
```

```
TSegInfo = record
  offSegment: Word;
  cbSegment: Word;
  flags: Word;
  cbAlloc: Word;
  h: THandle;
  alignShift: Word;
  reserved: array[0..1] of Word;
end;
```

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

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.

```
typedef struct tagSIZE {
    int cx;
    int cy;
} SIZE;
```

```
TSize = record
  cX: Integer;
  cY: Integer;
end;
```

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

The **STACKTRACEENTRY** structure contains information about one stack frame. This information enables an application to trace back through the stack of a specific task.

```
#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;
```

```
TStackTraceEntry=record
  dwSize: Longint;
  hTask: THandle;
  wSS: Word;
  wBP: Word;
  wCS: Word;
  wIP: Word;
  hModule: THandle;
  wSegment: Word;
  wFlags: Word;
end;
```

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

The **SYSHEAPINFO** structure contains information about the USER and GDI modules.

```
#include <toolhelp.h>

typedef struct tagSYSHEAPINFO {  /* shi */
    DWORD   dwSize;
    WORD    wUserFreePercent;
    WORD    wGDIFreePercent;
    HGLOBAL hUserSegment;
    HGLOBAL hGDISegment;
} SYSHEAPINFO;
```

```
TSysHeapInfo = record
  dwSize: Longint;
  wUserFreePercent: Word;
  wGDIFreePercent: Word;
  hUserSegment: THandle;
  hGDISegment: THandle;
end;
```

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

The **TASKENTRY** structure contains information about one task.

```
#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;
```

```
TTaskEntry = record
  dwSize: Longint;
  hTask: THandle;
  hTaskParent: THandle;
  hInst: THandle;
  hModule: THandle;
  wSS: Word;
  wSP: Word;
  wStackTop: Word;
  wStackMinimum: Word;
  wStackBottom: Word;
  wcEvents: Word;
  hQueue: THandle;
  szModule: array[0..max_Module_Name] of Char;
  wPSPOffset: Word;
  hNext: THandle;
end;
```

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**

TIMERINFO 3.1

The **TIMERINFO** structure contains the elapsed time since the current task became active and since the virtual machine (VM) started.

```
#include <toolhelp.h>

typedef struct tagTIMERINFO {  /* ti */
    DWORD dwSize;
    DWORD dwmsSinceStart;
    DWORD dwmsThisVM;
} TIMERINFO;
```

```
TTimerInfo = record
  dwSize: Longint;
  dwmsSinceStart: Longint;
  dwmsThisVM: Longint;
end;
```

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

The **TTPOLYCURVE** structure contains information about a curve in the outline of a TrueType character.

```
typedef struct tagTTPOLYCURVE {
    UINT    wType;
    UINT    cpfx;
    POINTFX apfx[1];
} TTPOLYCURVE;
```

```
TTTPolyCurve = record
  wType: Word;
  cpfx: Word;
  apfx: array[0..0] of TPointFX;
end;
```

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

The **TTPOLYGONHEADER** structure specifies the starting position and type of a TrueType character outline.

```
typedef struct tagTTPOLYGONHEADER {
    DWORD   cb;
    DWORD   dwType;
    POINTFX pfxStart;
} TTPOLYGONHEADER;
```

```
TPolygonHeader=record
  cb: Longint;
  dwType: Longint;
  pfxStart: TPointFX;
end;
```

Members

cb Specifies the number of bytes required by the **TTPOLYGONHEADER** structure.

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 character outline.

Comments The character outline is described by a series of **TTPOLYCURVE** structures that follow the **TTPOLYGONHEADER** structure.

See Also **POINTFX, TTPOLYCURVE**

VS_FIXEDFILEINFO 3.1

The **VS_FIXEDFILEINFO** structure contains version information about a file.

```
#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;
```

```
Tvs_FixedFileInfo=record
  dwSignature: Longint;          { e.g. $feef04bd }
  dwStrucVersion: Longint;       { e.g. $00000042 = "0.42" }
  dwFileVersionMS: Longint;      { e.g. $00030075 = "3.75" }
  dwFileVersionLS: Longint;      { e.g. $00000031 = "0.31" }
  dwProductVersionMS: Longint;   { e.g. $00030010 = "3.10" }
  dwProductVersionLS: Longint;   { e.g. $00000031 = "0.31" }
  dwFileFlagsMask: Longint;      { = $3F for version "0.42" }
  dwFileFlags: Longint;          { e.g. vff_Debug | vff_Prerelease }
  dwFileOS: Longint;                 { e.g. vos_DOS_Windows16 }
  dwFileType: Longint;           { e.g. vft_DRIVER }
  dwFileSubtype: Longint;            { e.g. vft2_DRV_Keyboard }
  dwFileDateMS: Longint;             { e.g. 0 }
  dwFileDateLS: Longint;             { e.g. 0 }
end;
```

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

The **WINDEBUGINFO** structure contains current system-debugging information for the debugging version of Windows 3.1.

```
typedef struct tagWINDEBUGINFO {
    UINT    flags;
    DWORD   dwOptions;
    DWORD   dwFilter;
    char    achAllocModule[8];
    DWORD   dwAllocBreak;
    DWORD   dwAllocCount;
} WINDEBUGINFO;
```

```
TWinDebugInfo = record
  Flags: Word;
  dwOptions: Longint;
  dwFilter: Longint;
  achAllocModule: array[0..7] of Char;
  dwAllocBreak: Longint;
  dwAllocCount: Longint;
end;
```

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.
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.

Constant	Value	Meaning
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.

Constant	Value	Meaning
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

The **WINDOWPLACEMENT** structure contains information about the placement of a window on the screen.

```
typedef struct tagWINDOWPLACEMENT {     /* wndpl */
    UINT  length;
    UINT  flags;
    UINT  showCmd;
    POINT ptMinPosition;
    POINT ptMaxPosition;
    RECT  rcNormalPosition;
} WINDOWPLACEMENT;
```

```
TWindowPlacement=record
  length: Word;
  flags: Word;
  showCmd: Word;
  ptMinPosition: TPoint;
  ptMaxPosition: TPoint;
  rcNormalPosition: TRect;
end;
```

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

The **WINDOWPOS** structure contains information about the size and position of a window.

```
typedef struct tagWINDOWPOS { /* wp */
    HWND    hwnd;
    HWND    hwndInsertAfter;
    int     x;
    int     y;
    int     cx;
    int     cy;
    UINT    flags;
} WINDOWPOS;
```

```
TWindowPos = record
  hWnd: HWnd;
  hWndInsertAfter: HWnd;
  x: Integer;
  y: Integer;
  cx: Integer;
  cy: Integer;
  flags: Word;
end;
```

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.

Value	Meaning
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).
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**

C H A P T E R

8

Macros

DECLARE_HANDLE 3.1

Syntax 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

Syntax #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.

Parameters *name* Specifies the name of the variable for which a pointer is created.

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

Syntax 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

Syntax BYTE GetBValue(rgb)

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

Syntax BYTE GetGValue(rgb)

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

Syntax BYTE GetRValue(rgb)

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**

MAKELP 3.1

Syntax void FAR* MAKELP(wSel, wOff)

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

Syntax LPARAM MAKELPARAM(wLow, wHigh)

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

Syntax LRESULT MAKELRESULT(wLow, wHigh)

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**

OFFSETOF 3.1

Syntax WORD OFFSETOF(lp)

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**

SELECTOROF 3.1

Syntax WORD SELECTOROF(lp)

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**

C H A P T E R

9

Printer escapes

MOUSETRAILS

Syntax 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.

Value	Meaning
–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.

POSTSCRIPT_DATA

The **POSTSCRIPT_DATA** printer escape is identical to the **PASSTHROUGH** escape.

POSTSCRIPT_IGNORE

Syntax 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.

SETALLJUSTVALUES

Syntax 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.

C H A P T E R

10

Dynamic Data Exchange transactions

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.

Value	Meaning
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**

C H A P T E R

11

Common dialog box messages

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 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 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 7, "Structures."

See Also **OpenFile, RegisterWindowMessage**

Index

A
ABC structure, 539
Advise transaction, DDEML, 59
Application (service) name, DDE servers, 40
Asynchronous transaction, DDEML, 61

B
BN_HILITE message, 536
BN_PAINT message, 536
BN_UNHILITE message, 536

C
CB_ADDSTRING message, 503
CB_DELETESTRING message, 504
CB_FINDSTRINGEXACT message, 505
CB_GETDROPPEDCONTROLRECT message, 505
CB_GETDROPPEDSTATE message, 506
CB_GETEXTENDEDUI message, 507
CB_GETITEMHEIGHT message, 507
CB_SETEXTENDEDUI message, 508
CB_SETITEMHEIGHT message, 509
CBN_CLOSEUP message, 537
CBN_SELENDCANCEL message, 537
CBN_SELENDOK message, 538
CBT_CREATEWND structure, 540
CBTACTIVATESTRUCT structure, 540
ChooseColor function, 8, 9
CHOOSECOLOR structure, 7, 8, 541
ChooseFont function, 11
CHOOSEFONT structure, 11, 544
Class Name Object command, OLE applications, 109
CLASSENTRY structure, 551
Client applications
  DDE transactions, 39
  OLE client applications
    asynchronous operations, 99
    Class Name Object command, 109
    closing, 110
    closing documents, 99
    compound documents, opening, 97
    copying objects, 103
    creating objects, 105
    DDE, direct use of, 124
    deleting objects, 103
    described, 80
    displaying objects, 102
    opening and closing objects, 103
    Paste and Paste Link commands, 107
    printing objects, 102
    saving documents, 99
    starting, 96
    Undo command, 108
Client user interface, OLE applications, 88
ClientCallback function, OLECLEINTVTBL structure, 618
Clipboard
  formats, 82
  OLE conventions, 81
Close function, OLESERVERDOCVTBL structure, 631
Color dialog box
  described, 3
  displaying basic colors, 7
  displaying custom colors, 8

HSL color model, 6
RGB color model, 5
COLOROKSTRING message, 719
CommDlgExtendedError function, 35
Common dialog box library
Color dialog box
described, 3
displaying basic colors, 7
displaying custom colors, 8
HSL color model, 6
RGB color model, 5
COMMDLG.DLL library, 1
common dialog boxes, described, 1
customizing common dialog boxes
described, 27
dialog box template, 31
displaying custom dialog boxes, 32
hook function, 28
error detection, 35
Find dialog box, 23, 26
Font dialog box, 11
Help button in common dialog boxes, 34
Open dialog box
displaying, 13
monitoring filenames, 19
monitoring list box controls, 18
Print dialog box, 20, 21
Print Setup dialog box, 20
Replace dialog box, 25, 26
Save As dialog box
displaying, 16
monitoring filenames, 19
monitoring list box controls, 18
Compound document, OLE applications
described, 72
illustrated, 72
opening, 97
COMSTAT structure, 552
CONVCONTEXT structure, 553
CONVINFO structure, 53, 554
Copy command
OLE client applications, 103
OLE server applications, 92, 115
CPLINFO structure, 557
Create function, OLESERVERVTBL structure, 638
CreateFromTemplate function, OLESERVERVTBL structure, 639
CTLINFO structure, 558
CTLSTYLE structure, 559
CTLTYPE structure, 561
Cut command
OLE client applications, 103
OLE server applications, 92, 115

D

Data handle
dynamic data exchange, 54
Data types, defined, 493
DdeAbandonTransaction function, 62
DdeAccessData function
command strings, 60
global memory objects, 56
DDEACK structure, 562
DdeAddData function, 57
DDEADVISE structure, 563
DdeCallback function, 44
DdeClientTransaction function
advise transaction, 59
execute transaction, 60
poke transaction, 58
request transaction, 57
synchronous and asynchronous transactions, 61
DdeConnect function, 49
DdeConnectList function, 52, 53
DdeCreateDataHandle function, 54
DdeCreateStringHandle function, 45
DDEDATA structure, 564
DdeDisconnect function, 52, 54
DdeDisconnectList function, 54
DdeEnableCallback function, 62
DdeFreeDataHandle function, 57
DdeFreeStringHandle function, 46
DdeGetData function, 56
DdeInitialize function
initializing DDEML, 42
monitoring DDE applications, 66

DdeKeepStringHandle function, 46
DdeNameService function, 47, 48
DDEPOKE structure, 565
DdePostAdvise function, 59
DdeQueryConvInfo function, 52, 53, 62
DdeQueryNextServer function, 53
DdeQueryString function, 45
DdeReconnect function, 52
DdeSetUserHandle function, 62
DdeUnaccessData function, 56
DdeUninitialize function, 43
DEBUGHOOKINFO structure, 566
DECLARE_HANDLE macro, 695
DECLARE_HANDLE32 macro, 695
DefLoadFromStream function, 123
DEVNAMES structure, 567
DllCreateFromClip function, 122
DllLoadFromStream function, 123
DOCINFO structure, 568
DoVerb function, OLEOBJECTVTBL structure, 625
DRIVERINFOSTRUCT structure, 569
DRVCONFIGINFO structure, 569
Dynamic data exchange (DDE)
  described, 37
  OLE libraries
    client applications, 124
    conversations, 128
    execute strings, 131, 132
    server applications, 127
    standard item names, 129
    System topic, items for, 128
    using for standard DDE operations, 77
Dynamic Data Exchange Management Library
  (DDEML)
  callback function, 44
  client and server interaction, 39
  conversations
    multiple conversations, 52
    single conversations, 49
    suspending, 62
    terminating, 43
  data management, 54
  described, 37
  error detection, 66
  initializing, 42
  item names, 40
  monitoring applications, 66
  vs. OLE, 76
  OLE, using with DDEML, 79
  service names
    described, 40
    registering, 47
    service-name filter, 48
  string management, 45
  System topic, 40
  topic names, 40
  transaction management
    advise transaction, 59
    asynchronous transactions, 61
    controlling transactions, 62
    execute transaction, 60
    poke transaction, 58
    request transaction, 57
    synchronous transactions, 61
    transaction classes, 63
    transaction summary, 64
  transaction, defined, 39

E

Edit function, OLESERVERVTBL structure, 640
EM_GETFIRSTVISIBLELINE message, 510
EM_GETPASSWORDCHAR message, 510
EM_GETWORDBREAKPROC message, 511
EM_SETREADONLY message, 511
EM_SETWORDBREAKPROC message, 512
Embedded object
  defined, 74
EnumClipboardFormats function, 108
Error detection
  common dialog boxes, 35
  DDEML functions, 66
EVENTMSG structure, 570
Execute function
  OLESERVERDOCVTBL structure, 636
  OLESERVERVTBL structure, 642

Execute strings, OLE
  international execute commands, 131
  required commands, 132
  syntax for standard commands, 131
Execute transaction, DDEML, 60
Exit function, OLESERVERVTBL structure, 641
ExtDeviceMode function, 21

F

FIELDOFFSET macro, 696
FILEOKSTRING message, 720
Find dialog box
  displaying, 23
  processing messages, 26
FINDMSGSTRING message, 26, 721
FINDREPLACE structure, 23, 25, 571
FindText function, 23
FIXED structure, 575
FMS_GETDRIVEINFO structure, 576
FMS_GETFILESEL structure, 577
FMS_LOAD structure, 578
Font dialog box, 11
Functions
  DdeCallback function, 44
  OLE functions
    asynchronous operations, 101
    document management, 98
    object creation, 105
    object handlers, 120
    server applications, 111

G

Get function, OLESTREAMVTBL structure, 644
GetBValue macro, 696
GetData function, 79
GetData function, OLEOBJECTVTBL structure, 626
GetGValue macro, 697
GetObject function, 79
GetObject function, OLESERVERDOCVTBL structure, 633
GetOpenFileName function, 13
GetRValue macro, 697
GetSaveFileName function, 16
GetWinFlags function
  initializing DDEML, 42
GLOBALENTRY structure, 579
GLOBALINFO structure, 582
GLYPHMETRICS structure, 583

H

HARDWAREHOOKSTRUCT structure, 584
Help button in common dialog boxes, 34
HELPMSGSTRING message, 721
HELPWININFO structure, 585
Hook function, common dialog boxes, 28
HSL color model, 6
HSZPAIR structure, 52, 585

I

Insert Object command, OLE applications, 106
Item name, DDE servers, 40

J

JUST_VALUE_STRUCT structure, 703

K

KERNINGPAIR structure, 586

L

LB_FINDSTRINGEXACT message, 513
LB_GETCARETINDEX message, 514
LB_SETCARETINDEX message, 514
LBN_SELCANCEL message, 538
LBSELCHSTRING message, 722
Linked object
  defined, 73
LoadString function, 16
LOCALENTRY structure, 587
LOCALINFO structure, 590
LOGFONT structure
  Font dialog box, 13
  TrueType fonts, server applications, 116

M

MAKELP macro, 697
MAKELPARAM macro, 698
MAKELRESULT macro, 698
MAT2 structure, 591
MEMMANINFO structure, 592
Metafile
  OLE server applications, 116
METAHEADER structure, 593
METARECORD structure, 594
MINMAXINFO structure, 595
MODULEENTRY structure, 596
MONCBSTRUCT structure, 597
MONCONVSTRUCT structure, 598
MONERRSTRUCT structure, 599
MONHSZSTRUCT structure, 600
Monitoring applications, 66
MONLINKSTRUCT structure, 602
MONMSGSTRUCT structure, 603
MOUSEHOOKSTRUCT structure, 604
MOUSETRAILS printer escape, 701

N

Native clipboard format, 82
NCCALCSIZE_PARAMS structure, 605
NEWCPLINFO structure, 606
NEWTEXTMETRIC structure, 607
NFYLOADSEG structure, 612
NFYLOGERROR structure, 613
NFYLOGPARAMERROR structure, 614
NFYRIP structure, 615
NFYSTARTDLL structure, 616

O

Object handler, OLE libraries
  creating objects in, 122
  described, 80
  implementing, 119
Object linking and embedding (OLE)
  benefits of OLE, 75
  client applications
    asynchronous operations, 99
    Class Name Object command, 109
    closing, 110
    closing documents, 99
    copying objects, 103
    creating objects, 105
    deleting objects, 103
    described, 95
    displaying objects, 102
    document management, 98
    opening and closing objects, 103
    Paste and Paste Link commands, 107
    printing objects, 102
    saving documents, 99
    starting, 96
    Undo command, 108
  compound documents
    described, 72
    illustrated, 72
    opening, 97
  data transfer
    client applications, 80
    client user interface, 88
    clipboard conventions, 81
    commands, new and changed, 88
    communication between libraries, 81
    object handlers, 80
    packages, 91
    registration database, 85
    server applications, 80
    server user interface, 92
    version control for servers, 87
  DDEML
    vs. OLE, 76
    using with OLE, 79
  dynamic data exchange
    client applications, 124
    conversations, 128
    DDE operations, using OLE for, 77
    execute strings, 131, 132
    server applications, 127
    standard item names, 129
    System topic, items for, 128
  embedded object, defined, 74
  formats for storing objects, 93
  linked object, defined, 73

object handlers
creating objects in, 122
implementing, 119
OLECLI.DLL library, 80
OLESVR.DLL library, 80
packages, 74
server applications
closing, 117
Cut and Copy commands, 115
functions, 111
opening documents or objects, 115
Save and Save As commands, 116
starting, 112
Update command, 116
verbs, 74
ObjectLink clipboard format, 82
ObjectLong function, OLEOBJECTVTBL structure, 627
OFFSETOF macro, 699
OleActivate function
Class Name Object command, implementing, 109
opening objects, 103
OleBlockServer function
asynchronous operations, 100
queued client-library requests, 114
OLECLIENT structure, 617
object handlers, 121
opening compound documents, 98
starting client applications, 96
OLECLIENTVTBL structure, 96, 617
OleClone function
copying objects to the clipboard, 105
restoring updated objects, 108
OleClose function, 103
OleCopyFromLink function, 106
OleCopyToClipboard function, 89, 103
OleCreate function, 106
OleCreateFromClip function
client applications, 107
object handlers, 122
OleCreateFromFile function, 78
OleCreateFromTemplate function, 106
OleCreateLinkFromClip function, 107
OleDelete function, 103
OleDraw function, 102
OleEnumFormats function, 103
OleGetData function, 110
OleGetLinkUpdateOptions command, 109
OleLoadFromStream function, 97
OLEOBJECT structure, 620
client applications, creating objects, 106
object handlers, 120
server applications
opening objects, 115
starting, 113
OleObjectConvert function, 110
OLEOBJECTVTBL structure, 120, 621
OleQueryBounds function, 102
OleQueryCreateFromClip function, 107
OleQueryLinkFromClip function, 107
OleQueryOpen function, 103
OleQueryReleaseError function
closing client applications, 110
creating objects, 106
OleQueryReleaseMethod function, 110
OleQueryReleaseStatus function
activating objects, 103
asynchronous operations, 101
closing client applications, 110
OleQuerySize function, 99
OleQueryType function, 107
OleReconnect function, 103
OleRegisterClientDoc function, 97, 105
OleRegisterServer function
DDE operations, 79
server applications, starting, 112
OleRegisterServerDoc function
DDE operations, 79
opening documents or objects, 115
starting server applications, 112
OleRelease function
closing client applications, 110
closing documents, 99
closing objects, 103
OleRenameServerDoc function, 116
OleRequestData function, 87
OleRevertClientDoc function, 99

OleRevokeClientDoc function, 99, 105
OleRevokeObject function, 118
OleRevokeServerDoc function, 118
OleSavedClientDoc function, 99, 105
OleSavedServerDoc function, 116
OleSaveToStream function, 99, 105
OLESERVER structure, 629
  object handlers, 121
  starting server applications, 112
OLESERVERDOC structure, 630
  object handlers, 121
  opening documents, 115
OLESERVERDOCVTBL structure, 630
  DDE operations, 79
  starting server applications, 112
OLESERVERVTBL structure, 636
  closing server applications, 117
  opening documents or objects, 115
  starting server applications, 112
  updating documents, 117
OleSetBounds function, 102
OleSetData function
  changing links, 110
  DDE operations, using OLE for, 78
  registering data formats, 87
OleSetHostNames function, 103
OleSetLinkUpdateOptions command, 109
OleSetTargetDevice function, 102
OLESTREAM structure, 643
  object handlers, 121
  opening compound documents, 98
  starting client applications, 96
OLESTREAMVTBL structure, 97, 98, 643
OLETARGETDEVICE structure, 645
OleUnblockServer function, 114
OleUpdate function
  displaying objects, 102
  updating links, 109
Open dialog box
  displaying, 13
  filenames, monitoring, 19
  list box controls, monitoring, 18
Open function, OLESERVERVTBL structure, 637
OPENFILENAME structure, 646
  Open dialog box, 13
  Save As dialog box, 16
OUTLINETEXTMETRIC structure, 655
OwnerLink clipboard format, 82

P

Package, OLE applications, 74, 91
PANOSE structure, 659
Paste command, OLE applications, 107
Paste Link command, OLE applications, 107
Paste Special command, OLE applications, 108
POINTFX structure, 664
Poke transaction, DDEML, 58
POSTSCRIPT_DATA printer escape
  See PASSTHROUGH printer escape
POSTSCRIPT_IGNORE printer escape, 702
Print dialog box, 20, 21
Print Setup dialog box, 20
PrintDlg function, 21
PRINTDLG structure, 21, 665
Printer
  default printer, 21
Put function, OLESTREAMVTBL structure, 644

R

RASTERIZER_STATUS structure, 673
RegisterClipboardFormat function, 87, 113
RegisterWindowMessage function
  Color dialog box, 10
  filenames, monitoring, 19
  Find and Replace dialog boxes, 26
  Help button in common dialog boxes, 34
  list box controls, monitoring, 18
  Open dialog box, 16
Registration database
  OLE applications, 85
Release function
  OLEOBJECTVTBL structure, 624
  OLESERVERDOCVTBL structure, 634
  OLESERVERVTBL structure, 641
Replace dialog box

displaying, 25
processing messages, 26
ReplaceText function, 25
Request transaction, DDEML, 57
RGB color model, 5

S

Save As command, OLE server applications, 116
Save As dialog box
displaying, 16
filenames, monitoring, 19
list box controls, monitoring, 18
Save command, OLE server applications, 116
Save function, OLESERVERDOCVTBL structure, 631
SearchFile function, 26
SEGINFO structure, 673
SELECTOROF macro, 699
Server applications
DDE transactions, 39
OLE servers
closing, 117
Cut and Copy commands, 115
DDE, direct use of, 127
DDE, required commands, 132
described, 80
functions, 111
opening documents or objects, 115
Save and Save As commands, 116
server user interface, 92
starting, 112
Update command, 116
version control, 87
Service name, DDE servers, 40
SETALLJUSTVALUES printer escape, 703
SetClipboardData function
client applications, 103
server applications, 115
SetColorScheme function
OLEOBJECTVTBL structure, 628
OLESERVERDOCVTBL structure, 635
SetData function, OLEOBJECTVTBL structure, 626
SetDocDimensions function, OLESERVERDOCVTBL structure, 633
SetHostNames function OLESERVERDOCVTBL structure, 632
SETRGBSTRING message, 723
SetTargetDevice function, OLEOBJECTVTBL structure, 627
SHAREVISTRING message, 723
Shell library
OLE applications, 85
Show function, OLEOBJECTVTBL structure, 624
SIZE structure, 675
STACKTRACEENTRY structure, 676
STM_GETICON message, 515
STM_SETICON message, 515
String handle, DDE, 45
Synchronous transaction, DDEML, 61
SYSHEAPINFO structure, 677
System topic, DDEML, 40
Systems topic, DDE-based OLE, 128

T

TASKENTRY structure, 678
Template, common dialog box, 31
TIMERINFO structure, 679
Topic name, DDE servers, 40
Transaction, DDE
defined, 39
TrueType fonts, server applications, 116
TTPOLYCURVE structure, 680
TTPOLYGONHEADER structure, 681

U

Undo command, OLE applications, 108
Update command, OLE server applications, 116

V

Verb, object linking and embedding, 74
Version control for OLE servers, 87
VS_FIXEDFILEINFO structure, 682

W

WINDEBUGINFO structure, 686
WINDOWPLACEMENT structure, 690
WINDOWPOS structure, 692
Windows data types, defined, 493
WinHelp function, 34
WM_CHOOSEFONT_GETLOGFONT message, 13, 516
WM_COMMNOTIFY message, 517
WM_DDE_ACK message, 517
WM_DDE_ADVISE message, 79, 520
WM_DDE_DATA message, 521
WM_DDE_EXECUTE message, 523
WM_DDE_INITIATE message, 524
WM_DDE_POKE message, 78, 526
WM_DDE_REQUEST message, 527
WM_DDE_TERMINATE message, 528
WM_DDE_UNADVISE message, 529
WM_DROPFILES message, 107, 530
WM_INITDIALOG message, 27
WM_PALETTEISCHANGING message, 530
WM_POWER message, 531
WM_QUEUESYNC message, 532
WM_SYSTEMERROR message, 532
WM_USER message, 533
WM_WINDOWPOSCHANGED message, 534
WM_WINDOWPOSCHANGING message, 535

X

XTYP_ADVDATA transaction, 705
XTYP_ADVREQ transaction, 706
XTYP_ADVSTART transaction, 59, 707
XTYP_ADVSTOP transaction, 60, 708
XTYP_CONNECT transaction, 49, 708
XTYP_CONNECT_CONFIRM transaction, 49, 52, 709
XTYP_DISCONNECT transaction, 52, 54, 710
XTYP_ERROR transaction, 711
XTYP_EXECUTE transaction, 60, 711
XTYP_MONITOR transaction, 68, 712
XTYP_POKE transaction, 58, 713
XTYP_REGISTER transaction, 47, 714
XTYP_REQUEST transaction, 45, 57, 715
XTYP_UNREGISTER transaction, 715
XTYP_WILDCONNECT transaction, 52, 716
XTYP_XACT_COMPLETE transaction, 61, 717

WINDOWS API
VOLUME III

BORLAND

Corporate Headquarters: 1800 Green Hills Road, P.O. Box 660001, Scotts Valley, CA 95067-0001, (408) 438-5300. Offices in: Australia, Belgium, Canada, Denmark, France, Germany, Hong Kong, Italy, Japan, Korea, Malaysia, Netherlands, New Zealand, Singapore, Spain, Sweden, Taiwan and United Kingdom ▪ Part #14MN-API03-31 ▪ BOR 3985