WINDOWS API
VOLUME I

REFERENCE GUIDE

BORLAND

Windows API Guide

Reference

Volume 1

Version 3.0
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

Copyright 1991, Borland International. All rights reserved.

Copyright © 1991 by Borland International. All rights reserved. All Borland products are trademarks or registered trademarks of Borland International, Inc. Other brand and product names are trademarks or registered trademarks of their respective holders.

PRINTED IN THE USA.
R1
10 9 8 7 6 5 4 3

C O N T E N T S

Introduction 1
Windows features 1
Window manager interface 2
Window manager interface function groups 3
Graphics device interface 3
Graphics device interface function groups 3
System services interface 4
System services interface function groups 4
Naming conventions 4
Parameter names 5
Windows calling convention 5
Manual overview 6
Volume 1 6
Volume 2 7
Document conventions 8
Other recommended reading 10
Windows functions 10

Part 1 Windows functions

Chapter 1 Window manager interface functions 13
Message functions 14
Generating and processing messages . 15
Translating messages 16
Examining messages 17
Sending messages 17
Avoiding message deadlocks 18
Window-creation functions 19
Window classes 20
System global classes 20
Application global classes 21
Application local classes 21
How Windows locates a class 21
How Windows determines the owner of a class 22
Registering a Window class 22
Shared Window classes 22
Predefined Window classes 22
Elements of a Window class 23
Class name 24
Window-function address 24
Instance handle 24
Class cursor 25
Class icon 25
Class background brush 25
Class menu 26
Class styles 27
Internal data structures 28
Window subclassing 28
Redrawing the client area 29
Class and private display contexts 29
Window function 30
Window messages 32
Default window function 33
Window styles 34
Overlapped windows 34
Owned windows 35
Pop-up windows 35
Child windows 35
Multiple document interface windows . 37
Title bar 38
System menu 38
Scroll bars 38
Menus 38
Window state 40
Life cycle of a window 40
Display and movement functions 42

Input functions 43
Hardware functions 43
Painting functions 44
How Windows manages the display .. 45
Display context types 46
Common display context 46
Class display context 47
Private display context 48
Window display context 49
Display-context cache 49
Painting sequence 50
WM_PAINT message 50
Update region 51
Window background 52
Brush alignment 52
Painting rectangular areas 53
Drawing icons 53
Drawing formatted text 54
Drawing gray text 56
Nonclient-area painting 57
Dialog box functions 57
Uses for dialog boxes 59
Modeless dialog box 59
Modal dialog box 59
System-modal dialog box 60
Creating a dialog box 60
Dialog box template 60
Dialog box measurements 61
Return values from a dialog box 61
Controls in a dialog box 62
Control identifiers 62
General control styles 62
Buttons 63
Edit controls 64
List boxes and directory listings 64
Combo boxes 65
Owner-draw dialog box controls ... 65
Messages for dialog box controls ... 66
Dialog box keyboard interface 66
Scrolling in dialog boxes 68
Scrolling functions 68
Standard scroll bars and scroll-bar controls 68
Scroll-bar thumb 69
Scrolling requests 70
Processing scroll messages 70
Scrolling the client area 71
Hiding a standard scroll bar 71
Menu functions 72
Information functions 73
System functions 73
Clipboard functions 74
Error functions 74
Caret functions 75
Creating and displaying a caret 75
Sharing the caret 76
Cursor functions 76
Pointing devices and the cursor 77
Displaying and hiding the cursor 77
Positioning the cursor 78
The cursor hotspot and confining the cursor 78
Creating a custom cursor 78
Hook functions 79
Filter-function chain 79
Installing a filter function 80
Property functions 80
Using property lists 81
Rectangle functions 82
Using rectangles in a Windows application 83
Rectangle coordinates 83
Creating and manipulating rectangles . 84
Chapter 2 Graphics device interface functions 87
Device-context functions 88
Device-context attributes 88
Saving a device context 90
Deleting a device context 90
Compatible device contexts 90
Information contexts 90
Drawing-tool functions 91
Drawing-tool uses 92
Brushes 92
Pens 93

Color 93
Color-palette functions 95
How color palettes work 96
Using a color palette 98
Drawing-attribute functions 99
Background mode and color 99
Stretch mode 100
Text color 100
Mapping functions 100
Constrained mapping modes 102
Partially constrained and unconstrained mapping modes 102
Partially constrained mapping mode 103
Unconstrained mapping mode 103
Transformation equations 103
Example: MM_TEXT 104
Example: MM_LOENGLISH 105
Coordinate functions 105
Region functions 106
Clipping functions 106
Line-output functions 107
Function coordinates 108
Pen styles, colors, widths 108
Ellipse and polygon functions 109
Function coordinates 109
Bounding rectangles 110
Bitmap functions 110
Bitmaps and devices 111
Device-independent bitmap functions 111
Text functions 112
Font functions 112
Font family 114
Character cells 115
Altering characters 116
Italic 116
Bold 116
Underline 116
Strikeout 116
Leading 116
Internal leading 117
External leading 117
Character set 117
ANSI character set 118
OEM character set 118
Symbol character set 118
Vendor-specific character sets 118
Pitch 118
Average character width 119
Maximum character width 119
Digitized aspect 119
Overhang 119
Selecting fonts with GDI 120
Font-mapping scheme 120
Example of font selection 123
Font files and font resources 124
Metafile functions 124
Creating a metafile 125
Storing a metafile in memory or on disk 126
Deleting a metafile 127
Changing how Windows plays a metafile 127
Printer-control functions 127
Printer-escape function 128
Creating output on a printer 128
Banding output 129
Starting and ending a print job 130
Terminating a print job 130
Information escapes 130
Additional escape calls 131
Environment functions 131
Chapter 3 System services interface functions 133
Module-management functions 134
Memory-management functions 134
Segment functions 136
Operating-system interrupt functions .. 137
Task functions 138
Resource-management functions 138
String-manipulation functions 139
Atom-management functions 140
Initialization-file functions 141
Communication functions 142

Sound functions 142
Utility macros and functions 143
File I/O functions 144
Debugging functions 144
Optimization-tool functions 145
Application-execution functions 145

Chapter 4 Functions directory 147
AccessResource 147
AddAtom 148
AddFontResource 148
AdjustWindowRect 149
AdjustWindowRectEx 150
AllocDStoCSAlias 150
AllocResource 151
AllocSelector 152
AnimatePalette 152
AnsiLower 153
AnsiLowerBuff 153
AnsiNext 154
AnsiPrev 154
AnsiToOem 154
AnsiToOemBuff 155
AnsiUpper 155
AnsiUpperBuff 156
AnyPopup 156
AppendMenu 156
AppendMenu 157
Arc 159
ArrangeIconicWindows 160
BeginDeferWindowPos 160
BeginPaint 161
BitBlt 162
BringWindowToTop 164
BuildCommDCB 164
CallMsgFilter 165
CallWindowProc 166
Catch 166
ChangeClipboardChain 167
ChangeMenu 167
ChangeSelector 168
CheckDlgButton 168
CheckMenuItem 169
CheckRadioButton 170
ChildWindowFromPoint 171
Chord 171
ClearCommBreak 172
ClientToScreen 172
ClipCursor 173
CloseClipboard 173
CloseComm 174
CloseMetaFile 174
CloseSound 174
CloseWindow 175
CombineRgn 175
CopyMetaFile 176
CopyRect 176
CountClipboardFormats 177
CountVoiceNotes 177
CreateBitmap 177
CreateBitmapIndirect 178
CreateBrushIndirect 179
CreateCaret 179
CreateCompatibleBitmap 180
CreateCompatibleDC 181
CreateCursor 182
CreateDC 182
CreateDialog 183
  Callback function 184
CreateDialogIndirect 185
  Callback function 186
CreateDialogIndirectParam 187
CreateDialogParam 188
CreateDIBitmap 189
CreateDIBPatternBrush 190
CreateDiscardableBitmap 191
CreateEllipticRgn 192
CreateEllipticRgnIndirect 192
CreateFont 193
CreateFontIndirect 195
CreateHatchBrush 196
CreateIC 196
CreateIcon 197
CreateMenu 198
CreateMetaFile 198
CreatePalette 199

CreatePatternBrush 199
CreatePen 200
CreatePenIndirect 200
CreatePolygonRgn 201
CreatePolyPolygonRgn 201
CreatePopupMenu 202
CreateRectRgn 203
CreateRectRgnIndirect 203
CreateRoundRectRgn 204
CreateSolidBrush 204
CreateWindow 205
CreateWindowEx 218
DebugBreak 219
DefDlgProc 220
DeferWindowPos 221
DefFrameProc 222
DefHookProc 224
DefineHandleTable 224
DefMDIChildProc 225
DefWindowProc 226
DeleteAtom 227
DeleteDC 227
DeleteMenu 228
DeleteMetaFile 229
DeleteObject 229
DestroyCaret 230
DestroyCursor 230
DestroyIcon 230
DestroyMenu 231
DestroyWindow 231
DeviceCapabilities 232
DeviceMode 235
DialogBox 235
  Callback Function 236
DialogBoxIndirect 237
  Callback Function 238
DialogBoxIndirectParam 239
DialogBoxParam 239
DispatchMessage 240
DlgDirList 241
DlgDirListComboBox 242
DlgDirSelect 244
DlgDirSelectComboBox 244
DOS3Call 245
DPtoLP 246
DrawFocusRect 247
DrawIcon 247
DrawMenuBar 248
DrawText 248
Ellipse 251
EmptyClipboard 252
EnableHardwareInput 252
EnableMenuItem 253
EnableWindow 254
EndDeferWindowPos 254
EndDialog 255
EndPaint 255
EnumChildWindows 256
  Callback function 257
EnumClipboardFormats 257
EnumFonts 258
  Callback function 258
EnumMetaFile 260
  Callback function 260
EnumObjects 261
  Callback function 262
EnumProps 262
  Fixed data segments 263
  Callback function 263
  Moveable data segments 264
  Callback function 264
EnumTaskWindows 265
  Callback function 265
EnumWindows 266
  Callback function 266
EqualRect 267
EqualRgn 267
Escape 268
EscapeCommFunction 269
ExcludeClipRect 269
ExcludeUpdateRgn 270
ExitWindows 271
ExtDeviceMode 271
ExtFloodFill 273
ExtTextOut 274
FatalAppExit 276

FatalExit 276
FillRect 277
FillRgn 278
FindAtom 278
FindResource 278
FindWindow 280
FlashWindow 280
FloodFill 281
FlushComm 282
_FPInit 282
_FPTerm 283
FrameRect 283
FrameRgn 284
FreeLibrary 284
FreeModule 285
FreeProcInstance 285
FreeResource 285
FreeSelector 286
GetActiveWindow 286
GetAspectRatioFilter 287
GetAsyncKeyState 287
GetAtomHandle 287
GetAtomName 288
GetBitmapBits 288
GetBitmapDimension 289
GetBkColor 289
GetBkMode 289
GetBrushOrg 290
GetBValue 290
GetCapture 290
GetCaretBlinkTime 291
GetCaretPos 291
GetCharWidth 291
GetClassInfo 292
GetClassLong 293
GetClassName 294
GetClassWord 294
GetClientRect 295
GetClipboardData 295
GetClipboardFormatName 296
GetClipboardOwner 297
GetClipboardViewer 297
GetClipBox 297
GetCodeHandle 298
GetCodeInfo 298
GetCommError 300
GetCommEventMask 301
GetCommState 301
GetCurrentPDB 302
GetCurrentPosition 302
GetCurrentTask 302
GetCurrentTime 303
GetCursorPos 303
GetDC 303
GetDCOrg 304
GetDesktopWindow 304
GetDeviceCaps 305
GetDialogBaseUnits 308
GetDIBits 309
GetDlgCtrlID 310
GetDlgItem 310
GetDlgItemInt 311
GetDlgItemText 312
GetDOSEnvironment 312
GetDoubleClickTime 313
GetDriveType 313
GetEnvironment 313
GetFocus 314
GetFreeSpace 315
GetGValue 316
GetInputState 316
GetInstanceData 316
GetKBCodePage 317
GetKeyboardState 317
GetKeyboardType 318
GetKeyNameText 319
GetKeyState 320
GetLastActivePopup 320
GetMapMode 321
GetMenu 321
GetMenuCheckMarkDimensions 321
GetMenuItemCount 322
GetMenuItemID 322
GetMenuState 322
GetMenuString 323
GetMessage 324

GetMessagePos 326
GetMessageTime 326
GetMetaFile 327
GetMetaFileBits 327
GetModuleFileName 327
GetModuleHandle 328
GetModuleUsage 328
GetNearestColor 329
GetNearestPaletteIndex 329
GetNextDlgGroupItem 329
GetNextDlgTabItem 330
GetNextWindow 330
GetNumTasks 331
GetObject 331
GetPaletteEntries 332
GetParent 333
GetPixel 333
GetPolyFillMode 334
GetPriorityClipboardFormat 334
GetPrivateProfileInt 335
GetPrivateProfileString 336
GetProcAddress 337
GetProfileInt 338
GetProfileString 338
GetProp 340
GetRgnBox 340
GetROP2 341
GetRValue 341
GetScrollPos 341
GetScrollRange 342
GetStockObject 343
GetStretchBltMode 344
GetSubMenu 345
GetSysColor 345
GetSysModalWindow 345
GetSystemDirectory 346
GetSystemMenu 346
GetSystemMetrics 347
GetSystemPaletteEntries 349
GetSystemPaletteUse 349
GetTabbedTextExtent 350
GetTempDrive 351
GetTempFileName 351
GetTextAlign 352
GetTextCharacterExtra 354
GetTextColor 354
GetTextExtent 354
GetTextFace 355
GetTextMetrics 355
GetThresholdEvent 356
GetThresholdStatus 356
GetTickCount 356
GetTopWindow 357
GetUpdateRect 357
GetUpdateRgn 358
GetVersion 359
GetViewportExt 359
GetViewportOrg 359
GetWindow 360
GetWindowDC 360
GetWindowExt 361
GetWindowLong 361
GetWindowOrg 362
GetWindowRect 362
GetWindowsDirectory 363
GetWindowTask 363
GetWindowText 364
GetWindowTextLength 364
GetWindowWord 365
GetWinFlags 365
GlobalAddAtom 366
GlobalAlloc 367
GlobalCompact 368
GlobalDeleteAtom 369
GlobalDiscard 369
GlobalDosAlloc 370
GlobalDosFree 370
GlobalFindAtom 371
GlobalFix 371
GlobalFlags 372
GlobalFree 372
GlobalGetAtomName 373
GlobalHandle 373
GlobalLock 374
GlobalLRUNewest 374
GlobalLRUOldest 375

GlobalNotify 375
Callback function 376
GlobalPageLock 376
GlobalPageUnlock 377
GlobalReAlloc 377
GlobalSize 379
GlobalUnfix 379
GlobalUnlock 380
GlobalUnWire 381
GlobalWire 381
GrayString 382
Callback function 383
HIBYTE 384
HideCaret 385
HiliteMenuItem 385
HIWORD 386
InflateRect 386
InitAtomTable 387
InSendMessage 387
InsertMenu 388
IntersectClipRect 391
IntersectRect 392
InvalidateRect 392
InvalidateRgn 393
InvertRect 394
InvertRgn 394
IsCharAlpha 395
IsCharAlphaNumeric 395
IsCharLower 395
IsCharUpper 396
IsChild 396
IsClipboardFormatAvailable 396
IsDialogMessage 397
IsDlgButtonChecked 398
IsIconic 398
IsRectEmpty 398
IsWindow 399
IsWindowEnabled 399
IsWindowVisible 399
IsZoomed 400
KillTimer 400
_lclose 401
_lcreat 401
LimitEmsPages 402
LineDDA 402
Callback function 403
LineTo 403
_llseek 404
LoadAccelerators 404
LoadBitmap 405
LoadCursor 406
LoadIcon 407
LoadLibrary 408
LoadMenu 409
LoadMenuIndirect 410
LoadModule 410
LoadResource 412
LoadString 412
LOBYTE 413
LocalAlloc 413
LocalCompact 414
LocalDiscard 415
LocalFlags 415
LocalFree 416
LocalHandle 416
LocalInit 416
LocalLock 417
LocalReAlloc 417
LocalShrink 419
LocalSize 420
LocalUnlock 420
LockData 420
LockResource 421
LockSegment 421
_lopen 422
LOWORD 423
LPtoDP 424
_lread 424
lstrcat 425
lstrcmp 425
lstrcmpi 426
lstrcpy 426
lstrlen 427
_lwrite 427
MAKEINTATOM 428
MAKEINTRESOURCE 429

MAKELONG 429
MAKEPOINT 429
MakeProcInstance 429
MapDialogRect 430
MapVirtualKey 431
max 432
MessageBeep 432
MessageBox 432
min 434
ModifyMenu 435
MoveTo 438
MoveWindow 438
MulDiv 439
NetBIOSCall 440
OemKeyScan 440
OemToAnsi 441
OemToAnsiBuff 442
OffsetClipRgn 442
OffsetRect 443
OffsetRgn 443
OffsetViewportOrg 444
OffsetWindowOrg 444
OpenClipboard 445
OpenComm 445
OpenFile 446
OpenIcon 449
OpenSound 449
OutputDebugString 449
PaintRgn 450
PALETTEINDEX 450
PALETTERGB 450
PatBlt 451
PeekMessage 452
Pie 454
PlayMetaFile 455
PlayMetaFileRecord 455
Polygon 456
Polyline 456
PolyPolygon 457
PostAppMessage 458
PostMessage 458
PostQuitMessage 459
ProfClear 459
ProfFinish 460
ProfFlush 460
ProfInsChk 460
ProfSampRate 461
ProfSetup 462
ProfStart 462
ProfStop 462
PtInRect 463
PtInRegion 463
PtVisible 463
ReadComm 464
RealizePalette 465
Rectangle 465
RectInRegion 466
RectVisible 466
RegisterClass 467
Callback function 467
RegisterClipboardFormat 468
RegisterWindowMessage 468
ReleaseCapture 469
ReleaseDC 469
RemoveFontResource 470
RemoveMenu 470
RemoveProp 471
ReplyMessage 472
ResizePalette 473
RestoreDC 473
RGB 474
RoundRect 474
SaveDC 476
ScaleViewportExt 476
ScaleWindowExt 477
ScreenToClient 477
ScrollDC 478
ScrollWindow 479
SelectClipRgn 480
SelectObject 481
SelectPalette 483
SendDlgItemMessage 483
SendMessage 484
SetActiveWindow 485
SetBitmapBits 485
SetBitmapDimension 486

SetBkColor 486
SetBkMode 487
SetBrushOrg 487
SetCapture 488
SetCaretBlinkTime 488
SetCaretPos 488
SetClassLong 489
SetClassWord 490
SetClipboardData 491
SetClipboardViewer 493
SetCommBreak 494
SetCommEventMask 494
SetCommState 495
SetCursor 495
SetCursorPos 496
SetDIBits 496
SetDIBitsToDevice 498
SetDlgItemInt 499
SetDlgItemText 500
SetDoubleClickTime 500
SetEnvironment 501
SetErrorMode 501
SetFocus 502
SetHandleCount 502
SetKeyboardState 503
SetMapMode 503
SetMapperFlags 505
SetMenu 505
SetMenuItemBitmaps 506
SetMessageQueue 507
SetMetaFileBits 507
SetPaletteEntries 508
SetParent 508
SetPixel 509
SetPolyFillMode 509
SetProp 510
SetRect 511
SetRectEmpty 511
SetRectRgn 512
SetResourceHandler 512
  Callback function 513
SetROP2 514
SetScrollPos 515
SetScrollRange 516
SetSoundNoise 516
SetStretchBltMode 517
SetSwapAreaSize 518
SetSysColors 519
SetSysModalWindow 520
SetSystemPaletteUse 520
SetTextAlign 522
SetTextCharacterExtra 523
SetTextColor 523
SetTextJustification 524
SetTimer 525
  Callback function 526
SetViewportExt 526
SetViewportOrg 527
SetVoiceAccent 528
SetVoiceEnvelope 529
SetVoiceNote 530
SetVoiceQueueSize 531
SetVoiceSound 531
SetVoiceThreshold 532
SetWindowExt 532
SetWindowLong 533
SetWindowOrg 534
SetWindowPos 535
SetWindowsHook 536
    WH_CALLWNDPROC 538
    WH_GETMESSAGE 539
    WH_JOURNALPLAYBACK 540
    WH_JOURNALRECORD 541
    WH_KEYBOARD 542
    WH_MSGFILTER 543
    WH_SYSMSGFILTER 544
SetWindowText 545
SetWindowWord 545
ShowCaret 546
ShowCursor 546
ShowOwnedPopups 547
ShowScrollBar 547
ShowWindow 548
SizeofResource 549
StartSound 549
StopSound 550

StretchBlt 550
StretchDIBits 552
SwapMouseButton 554
SwapRecording 554
SwitchStackBack 555
SwitchStackTo 555
SyncAllVoices 556
TabbedTextOut 556
TextOut 557
Throw 558
ToAscii 559
TrackPopupMenu 560
TranslateAccelerator 560
TranslateMDISysAccel 562
TranslateMessage 562
TransmitCommChar 563
UngetCommChar 563
UnhookWindowsHook 564
UnionRect 565
UnlockData 565
UnlockResource 565
UnlockSegment 566
UnrealizeObject 566
UnregisterClass 567
UpdateColors 568
UpdateWindow 568
ValidateCodeSegments 568
ValidateFreeSpaces 569
ValidateRect 569
ValidateRgn 570
VkKeyScan 570
WaitMessage 571
WaitSoundState 572
WindowFromPoint 572
WinExec 573
WinHelp 574
WriteComm 576
WritePrivateProfileString 577
WriteProfileString 578
wsprintf 579
wvsprintf 581
Yield 583

Part 2 Windows messages

Chapter 5 Messages overview 587
Window-management messages 587
Initialization messages 589
Input messages 589
System messages 590
Clipboard messages 591
System information messages 592
Control messages 592
  Button-control messages 593
  Edit-control messages 593
  List-box messages 594
  Combo-box messages 595
  Owner draw-control messages 596
Notification messages 597
  Button notification codes 597
  Edit-control notification codes 597
  List-box notification codes 598
  Combo-box notification codes 598
Scroll-bar messages 598
Nonclient-area messages 598
Multiple document interface messages . 600

Chapter 6 Messages directory 601
BM_GETCHECK 603
BM_GETSTATE 603
BM_SETCHECK 603
BM_SETSTATE 604
BM_SETSTYLE 604
BN_CLICKED 605
BN_DOUBLECLICKED 606
CB_ADDSTRING 606
CB_DELETESTRING 606
CB_DIR 607
CB_FINDSTRING 607
CB_GETCOUNT 608
CB_GETCURSEL 608
CB_GETEDITSEL 608
CB_GETITEMDATA 609
CB_GETLBTEXT 609
CB_GETLBTEXTLEN 609
CB_INSERTSTRING 610
CB_LIMITTEXT 610

CB_RESETCONTENT 610
CB_SELECTSTRING 611
CB_SETCURSEL 611
CB_SETEDITSEL 612
CB_SETITEMDATA 612
CB_SHOWDROPDOWN 612
CBN_DBLCLK 613
CBN_DROPDOWN 613
CBN_EDITCHANGE 613
CBN_EDITUPDATE 614
CBN_ERRSPACE 614
CBN_KILLFOCUS 614
CBN_SELCHANGE 615
CBN_SETFOCUS 615
DM_GETDEFID 615
DM_SETDEFID 615
EM_CANUNDO 616
EM_EMPTYUNDOBUFFER 616
EM_FMTLINES 616
EM_GETHANDLE 617
EM_GETLINE 617
EM_GETLINECOUNT 617
EM_GETMODIFY 618
EM_GETRECT 618
EM_GETSEL 618
EM_LIMITTEXT 618
EM_LINEFROMCHAR 619
EM_LINEINDEX 619
EM_LINELENGTH 619
EM_LINESCROLL 620
EM_REPLACESEL 620
EM_SETHANDLE 620
EM_SETMODIFY 621
EM_SETPASSWORDCHAR 621
EM_SETRECT 621
EM_SETRECTNP 622
EM_SETSEL 622
EM_SETTABSTOPS 622
EM_SETWORDBREAK 623
  Callback Function 623
EM_UNDO 624
EN_CHANGE 624
EN_ERRSPACE 625
EN_HSCROLL 625
EN_KILLFOCUS 625
EN_MAXTEXT 626
EN_SETFOCUS 626
EN_UPDATE 626
EN_VSCROLL 627
LB_ADDSTRING 627
LB_DELETESTRING 627
LB_DIR 628
LB_FINDSTRING 628
LB_GETCARETINDEX 629
LB_GETCOUNT 629
LB_GETCURSEL 629
LB_GETHORIZONTALEXTENT 630
LB_GETITEMDATA 630
LB_GETITEMHEIGHT 630
LB_GETITEMRECT 631
LB_GETSEL 631
LB_GETSELCOUNT 631
LB_GETSELITEMS 631
LB_GETTEXT 632
LB_GETTEXTLEN 632
LB_GETTOPINDEX 632
LB_INSERTSTRING 633
LB_RESETCONTENT 633
LB_SELECTSTRING 633
LB_SELITEMRANGE 634
LB_SETCARETINDEX 634
LB_SETCOLUMNWIDTH 635
LB_SETCURSEL 635
LB_SETHORIZONTALEXTENT 635
LB_SETITEMDATA 636
LB_SETITEMHEIGHT 636
LB_SETSEL 636
LB_SETTABSTOPS 637
LB_SETTOPINDEX 637
LBN_DBLCLK 638
LBN_ERRSPACE 638
LBN_KILLFOCUS 638
LBN_SELCHANGE 639
LBN_SETFOCUS 639
WM_ACTIVATE 639
WM_ACTIVATEAPP 640

WM_ASKCBFORMATNAME 640
WM_CANCELMODE 641
WM_CHANGECBCHAIN 641
WM_CHAR 641
WM_CHARTOITEM 642
WM_CHILDACTIVATE 643
WM_CLEAR 643
WM_CLOSE 643
WM_COMMAND 644
WM_COMPACTING 644
WM_COMPAREITEM 645
WM_COPY 645
WM_CREATE 646
WM_CTLCOLOR 646
WM_CUT 647
WM_DEADCHAR 647
WM_DELETEITEM 648
WM_DESTROY 648
WM_DESTROYCLIPBOARD 649
WM_DEVMODECHANGE 649
WM_DRAWCLIPBOARD 649
WM_DRAWITEM 650
WM_ENABLE 650
WM_ENDSESSION 650
WM_ENTERIDLE 651
WM_ERASEBKGND 651
WM_FONTCHANGE 652
WM_GETDLGCODE 652
WM_GETFONT 653
WM_GETMINMAXINFO 653
WM_GETTEXT 654
WM_GETTEXTLENGTH 654
WM_HSCROLL 655
WM_HSCROLLCLIPBOARD 656
WM_ICONERASEBKGND 656
WM_INITDIALOG 657
WM_INITMENU 657
WM_INITMENUPOPUP 658
WM_KEYDOWN 658
WM_KEYUP 659
WM_KILLFOCUS 660
WM_LBUTTONDBLCLK 660
WM_LBUTTONDOWN 661
WM_LBUTTONUP 661
WM_MBUTTONDBLCLK 662
WM_MBUTTONDOWN 662
WM_MBUTTONUP 663
WM_MDIACTIVATE 663
WM_MDICASCADE 664
WM_MDICREATE 664
WM_MDIDESTROY 665
WM_MDIGETACTIVE 665
WM_MDIICONARRANGE 666
WM_MDIMAXIMIZE 666
WM_MDINEXT 666
WM_MDIRESTORE 667
WM_MDISETMENU 667
WM_MDITILE 667
WM_MEASUREITEM 668
WM_MENUCHAR 668
WM_MENUSELECT 669
WM_MOUSEACTIVATE 669
WM_MOUSEMOVE 670
WM_MOVE 671
WM_NCACTIVATE 671
WM_NCCALCSIZE 671
WM_NCCREATE 672
WM_NCDESTROY 672
WM_NCHITTEST 672
WM_NCLBUTTONDBLCLK 673
WM_NCLBUTTONDOWN 674
WM_NCLBUTTONUP 674
WM_NCMBUTTONDBLCLK 674
WM_NCMBUTTONDOWN 675
WM_NCMBUTTONUP 675
WM_NCMOUSEMOVE 675
WM_NCPAINT 676
WM_NCRBUTTONDBLCLK 676
WM_NCRBUTTONDOWN 676
WM_NCRBUTTONUP 677
WM_NEXTDLGCTL 677
WM_PAINT 677
WM_PAINTCLIPBOARD 678
WM_PAINTICON 678
WM_PALETTECHANGED 679
WM_PARENTNOTIFY 679

WM_PASTE 680
WM_QUERYDRAGICON 680
WM_QUERYENDSESSION 681
WM_QUERYNEWPALETTE 681
WM_QUERYOPEN 681
WM_QUIT 682
WM_RBUTTONDBLCLK 682
WM_RBUTTONDOWN 682
WM_RBUTTONUP 683
WM_RENDERALLFORMATS 683
WM_RENDERFORMAT 684
WM_SETCURSOR 684
WM_SETFOCUS 684
WM_SETFONT 685
WM_SETREDRAW 685
WM_SETTEXT 686
WM_SHOWWINDOW 686
WM_SIZE 687
WM_SIZECLIPBOARD 687
WM_SPOOLERSTATUS 688
WM_SYSCHAR 688
WM_SYSCOLORCHANGE 689
WM_SYSCOMMAND 690
WM_SYSDEADCHAR 691
WM_SYSKEYDOWN 691
WM_SYSKEYUP 693
WM_TIMECHANGE 694
WM_TIMER 694
WM_UNDO 695
WM_VKEYTOITEM 695
WM_VSCROLL 695
WM_VSCROLLCLIPBOARD 696
WM_WININICHANGE 697
Index 699

TABLES

0.1: Standard prefixes 5
0.2: Document conventions 8
0.3: Windows API guide 9
1.1: Window class elements 23
1.2: Window class styles 27
1.3: Default actions for messages 33
1.4: Defaults for a display context 46
1.5: Drawing format styles 54
1.6: Control characters and actions 55
1.7: Dialog box controls 65
1.8: Dialog box keyboard interface 67
2.1: Default device-context attributes and related GDI functions 89
2.2: Font-mapping characteristics 121
4.1: Raster operations 163
4.2: Control classes 207
4.3: Window styles 209
4.4: Control styles 211
4.5: Extended window styles 219
4.6: DOS file attributes 242
4.7: DrawText formats 250
4.8: Communications error codes 300
4.9: GDI information indexes 305
4.10: System metric indexes 348
4.11: Message box types 433
4.12: Raster operations 452
4.13: Predefined data formats 492
4.14: Event values 494
4.15: Mapping modes 504
4.16: Drawing modes 514
4.17: System color indexes 519
4.18: Window states 548
6.1: Button styles 604
6.2: Hit-test codes 673

FIGURES

1.1: Caret shapes 76
1.2: Property list 81
1.3: Rectangle limits 84
1.4: Intersection of two rectangles 85
1.5: Union of two rectangles 85
2.1: Information flow to an output device . 88
2.2: Hatched brush patterns 92
2.3: Pen patterns 93
2.4: Palette manager color-mapping algorithm 97
2.5: Mapping with MM_TEXT 104
2.6: Mapping with MM_LOENGLISH . . . 105
2.7: Arc and its bounding rectangle 108
2.8: Styled-Pen and Solid-Pen Rectangles 109
2.9: Fonts from two typefaces 113
2.10: Cross-stroke and stem 114
2.11: Serifs 114
2.12: Character-cell dimensions 115
2.13: Strikeout characters 116
2.14: Internal leading 117
2.15: External leading 117
2.16: A GDI font table 120
2.17: Sample font selection ratings 123

INTRODUCTION

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

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

Windows features

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

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

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

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

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

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

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

Window manager interface

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

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

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

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

Window manager interface function groups

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

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

Graphics device interface

The graphics device interface (GDI) contains the functions that perform device-independent graphic operations within a Windows application. These functions create a wide variety of line, text, and bitmap output on a number of different output devices. GDI allows an application to create pens, brushes, fonts, and bitmaps for specific output operations.

Graphics device interface function groups

The following list describes the function groups found in GDI:

- Device-context functions
- Ellipse and polygon functions
- Drawing-tool functions
- Bitmap functions
- Drawing-attribute functions
- Text functions
- Mapping functions
- Font functions
- Coordinate functions

- Metafile functions
- Region functions
- Printer-escape functions
- Clipping functions
- Environment functions
- Line-output functions
- System functions

System services interface

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

System services interface function groups

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

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

Naming conventions

Many Windows functions have been named with a verb-noun model to help you remember and become familiar with the function. The function name indicates both what the function does (verb) and the target of its action (noun). All function names begin with an uppercase letter. If the name is composed of several words, each word begins with an uppercase letter and all words

are adjoined (no spaces or underscore characters separate the words). Some examples of function names are shown below:

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

Parameter names

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

Table 0.1
Standard prefixes

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

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

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

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

Windows calling convention

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

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

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

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

Manual overview

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

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

Volume 1

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

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

Chapter 2, "Graphics device interface functions," categorizes the functions that perform device-independent graphics operations in the Windows environment, provides brief descriptions of the

functions, and explains the most important features of the Windows graphics interface.

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

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

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

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

Volume 2

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

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

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

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

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

Chapter 11, "Binary and ternary raster-operation codes," describes the raster operations used for line output and those used for bitmap output.

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

Chapter 13, "Windows DDE protocol definition," contains an alphabetical listing and description of the Windows messages which comprise the Windows Dynamic Data Exchange protocol.

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

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

Document conventions

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

The following document conventions are used throughout this manual:

Table 0.2
Document conventions

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

Table 0.2: Document conventions (continued)

	SetCursorPos function indicates that you must substitute values for the *X* and *Y* coordinates, separated by a comma: **SetCursorPos(*X, Y*)**
`Monospaced type`	Code examples are displayed in a nonproportional typeface.
⋮	Vertical ellipses in program examples indicate that a portion of the program is omitted.
...	Ellipses following an item indicate that more items having the same form may appear. In the following example, the horizontal ellipses indicate that you can specify more than one *breakaddress* for the **g** command: **g** [[=*startaddress*]] [[*breakaddress*]]...
[[ ]]	Double brackets enclose optional fields or parameters in command lines and syntax statements. In the following example, *option* and *executable-file* are optional parameters of the **RC** command: **RC** [[*option*]] *filename* [[*executable-file*]]
|	A vertical bar indicates that you may enter one of the entries shown on either side of the bar. The following command-line syntax illustrates the use of a vertical bar: **DB** [[*address* | *range*]] The bar indicates that following the **Dump Bytes** command (**DB**), you can specify either an *address* or a *range*.
" "	Quotation marks set off terms defined in the text.
{ }	Curly braces indicate that you must specify one of the enclosed items.
SMALL CAPITAL LETTERS	Small capital letters indicate the names of keys and key sequences, such as: ALT + SPACEBAR
3.0	A Microsoft Windows version number indicates that a function, message, or data structure is compatible only with the specified version and later versions.

Table 0.3
Windows API guide

Title	**Contents**
Reference	Is a comprehensive guide to all the details of the Microsoft Windows application program interface (API). The *Reference* lists in alphabetical order all the current functions, messages, and data structures of the API, and provides extensive overviews on how to use the API.

The *Windows API guide* will answer many of your programming questions. This book provides information on each Windows application programming interface (API) and describes its calls and services.

Other recommended reading

The following books are recommended for efficient Windows programming:

Programming Windows. Charles Petzold. 862 pages, softcover. An updated second edition will be available in October 1990.

Windows: Programmer's Problem Solver. Richard Wilton. 400 pages, softcover. Available November 1990.

Microsoft C Run-Time Library Reference. Covers version 6. Microsoft Corporation. 852 pages, softcover.

P A R T

1

Windows functions

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

C H A P T E R

1

Window manager interface functions

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

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

Message functions

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

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

Generating and processing messages

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

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

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

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

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

Once a main function has a message from a queue, it can dispatch the message to a window function by using the **DispatchMessage** function. This function directs Windows to call the window function of the window associated with the message, and then passes the content of the message as function arguments. The

window function can then process the message and carry out any requested changes to the window. When the window function returns, Windows returns control to the main function. The main function can then pull the next message from the queue.

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

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

Translating messages

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

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

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

Applications that use accelerator keys must load an accelerator table from the resource file by using the **LoadAccelerator** function, and then translate

keyboard messages into accelerator-key messages by using the **Translate-Accelerator** function. The main loop for applications that use accelerator keys should have the following form:

```
while (GetMessage((LPMSG)&msg, (HWND)NULL, 0, 0))
    {
    if (TranslateAccelerator(hWindow, hAccel, ((LPMSG)&msg) == 0)
     {
      TranslateMessage((LPMSG)&msg);
      DispatchMessage((LPMSG)&msg);
     }
    }
exit(msg.wParam);
```

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

Examining messages

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

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

Sending messages

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

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

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

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

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

Avoiding message deadlocks

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

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

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

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

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

Window-creation functions

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

Function	Description
AdjustWindowRect	Computes the size of a window to fit a given client area.
AdjustWindowRectEx	Computes the size of a window with extended style to fit a given client area.
CreateWindow	Creates overlapped, pop-up, and child windows.
CreateWindowEx	Creates overlapped, pop-up, and child windows with extended styles.
DefDlgProc	Provides default processing for those dialog-box messages that an application does not process.
DefFrameProc	Provides default processing for those multiple document interface (MDI) frame window messages that an application does not process.
DefMDIChildProc	Provides default processing those for MDI child window messages an that application does not process.

DefWindowProc	Provides default processing for those window messages that an DefWindowProc function
DestroyWindow	Destroys a window.
GetClassInfo	Retrieves information about a specified class.
GetClassLong	Retrieves window-class information from a **WNDCLASS** structure.
GetClassName	Retrieves a window-class name.
GetClassWord	Retrieves window-class information from a **WNDCLASS** structure.
GetLastActivePopup	Determines which popup window owned by another window was most recently active.
GetWindowLong	Retrieves information about a window.
GetWindowWord	Retrieves information about a window.
RegisterClass	Registers a window class.
SetClassLong	Replaces information in a **WNDCLASS** structure.
SetClassWord	Replaces information in a **WNDCLASS** structure.
SetWindowLong	Changes a window attribute.
SetWindowWord	Changes a window attribute.
UnregisterClass	Removes a window class from the window-class table.

Window classes

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

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

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

System global classes

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

Application global classes

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

Application local classes

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

How Windows locates a class

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

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

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

How Windows determines the owner of a class

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

Registering a Window class

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

Shared Window classes

See "Application global classes," on page 21 for more information.

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

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

Predefined Window classes

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

Elements of a Window class

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

Table 1.1 shows the window class elements.

Table 1.1
Window class elements

Element	Purpose
Class name	Distinguishes the class from other registered classes.
Window-function address	Points to the function that processes all messages that are sent to windows in the class, and defines the behavior of the window.
Instance handle	Identifies the application that registered the class.
Class cursor	Defines the shape of the cursor when the cursor is in a window of the class.
Class icon	Defines the shape of the icon Windows displays when a window belonging to the class is closed.
Class background brush	Defines the color and pattern Windows uses to fill the client area when the window is opened or painted.
Class menu	Specifies the default menu used for any window in the class that does not explicitly define a menu.
Class styles	Defines how to update the window after moving or resizing, how to process double-clicks of the mouse, how to allocate space for the display context, and other aspects of the window.
Class extra	Specifies the amount of memory (in bytes) that Windows should reserve at the end of the class data structure.
Window extra	Specifies the amount of memory (in bytes) that Windows should reserve at the end of any window structure an application creates with this class.

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

Class name

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

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

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

Window-function address

See Chapter 10, "Module-definition statements," in Reference, Volume 2, for more information on exporting functions. For details about the window function, see page 30.

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

Instance handle

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

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

Class cursor

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

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

Class icon

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

Windows does not require a class icon. If a class icon is not defined, Windows assumes the application will draw the icon whenever the window is minimized. In this case, Windows sends appropriate messages to the window procedure, requesting that the icon be painted.

Class background brush

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

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

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

- COLOR_ACTIVECAPTION
- COLOR_APPWORKSPACE

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

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

Class menu

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

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

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

Class styles

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

Table 1.2
Window class styles

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

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

Internal data structures

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

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

The following section describes some ways in which a window class or window can be modified.

Window subclassing

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

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

Similarly, the **SetClassLong** function changes the window function associated with a window class. Any window that is subsequently created with that class will be associated with the replacement window function for that class, as will the window whose handle is passed to **SetClassLong**. Other existing windows

that were previously created with the class are not affected, however.

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

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

Redrawing the client area

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

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

Class and private display contexts

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

If no explicit display-context style is given, Windows assumes that each window will use a display context retrieved from a pool of contexts maintained by Windows. In such cases, each window

must retrieve and initialize the display context before painting, and then free it after painting.

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

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

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

Window function

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

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

windows, such as losing access to the keyboard or becoming the active window.

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

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

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

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

A window function also receives messages that are really intended to be processed by the system. These messages, called nonclient-area messages, inform the function either that the user

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

Window messages

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

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

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

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

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

Windows expects to make an intersegment call to the window function, so the function must be declared with the **FAR** attribute.

The window-function name must be exported by including it in an **EXPORTS** statement in the application's module-definition file.

Default window function

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

Table 1.3
Default actions for messages

Message	Default Action
WM_ACTIVATE	Sets or kills the input focus.
WM_CANCELMODE	Terminates internal processing of standard scroll bar input, terminates internal menu processing, and releases mouse capture.
WM_CLOSE	Calls the **DestroyWindow** function.
WM_CTLCOLOR	Sets the background and text color and returns a handle to the brush used to fill the control background.
WM_ERASEBKGND	Fills the client area with the color and pattern specified by the class brush, if any.
WM_GETTEXT	Copies the window title into a specified buffer.
WM_GETTEXTLENGTH	Returns the length (in characters) of the window title.
WM_ICONERASEBKGND	Fills the icon client area with the background brush of the parent window.
WM_NCACTIVATE	Activates or deactivates the window and draws the icon or title bar to show the new state.
WM_NCCALCSIZE	Computes the size of the client area.
WM_NCCREATE	Initializes standard scroll bars, if any, and sets the default title for the window.
WM_NCDESTROY	Frees any space internally allocated for the window title.
WM_NCHITTEST	Determines what part of the window the mouse is in.
WM_NCLBUTTONDBLCLK	Tests the given point to determine the location of the mouse and, if necessary, generates additional messages.
WM_NCLBUTTONDOWN	Determines whether the left mouse button was pressed while the mouse was in the nonclient area of a window.

Table 1.3: Default actions for messages (continued)

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

Window styles

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

Overlapped windows

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

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

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

Owned windows

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

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

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

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

Pop-up windows

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

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

Child windows

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

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

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

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

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

For information about mapping, see "Mapping functions" on page 100.

Windows always positions the child window relative to the upper left corner of the parent window's client area. The coordinates are always client coordinates. If all or part of a child window is moved outside the visible portion of the parent window's client area, the child window is clipped; that is, the portion outside the parent window's client area is not displayed.

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

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

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

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

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

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

Multiple document interface windows

Windows multiple document interface (MDI) provides applications with a standard interface for displaying multiple documents within the same instance of an application. An MDI application creates a frame window which contains a client window in place of its client area. An application creates an MDI client window by calling **CreateWindow** with the class MDICLIENT and passing a **CLIENTCREATESTRUCT** data structure as the function's *lpParam* parameter. This client window

in turn can own multiple child windows, each of which displays a separate document. An MDI application controls these child windows by sending messages to its client window.

Title bar

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

System menu

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

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

Scroll bars

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

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

Menus

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

to indicate which command was selected. Windows provides two types of menus: menu bars (sometimes called static menus) and pop-up menus.

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

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

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

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

When the application calls **AppendMenu, InsertMenu**, or **ModifyMenu** to add an owner-draw menu item to a menu or to change an existing menu item to be an owner-draw menu item, the application can supply a 32-bit value as the *lpNewItem* parameter to the function. The application can use this value to maintain additional data associated with the item. This value is

available to the application as the **itemData** field of the structures pointed to by the *lParam* parameter of the WM_MEASUREITEM and WM_DRAWITEM messages. For example, if an application were to draw the text in a menu item using a specific color, the 32-bit value could contain a pointer to a string. The application could then set the text color before drawing the item when it received the WM_DRAWITEM message.

Window state

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

- WS_DISABLED
- WS_MINIMIZE
- WS_MAXIMIZE
- WS_VISIBLE

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

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

Life cycle of a window

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

The first step in a window's life cycle is creation. Given a registered window class with a corresponding window function, the application uses the **CreateWindow** function to create the window. This function directs Windows to prepare internal data

structures for the window and to return a unique integer value, called a window handle, that the application can use to identify the window in subsequent function calls.

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

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

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

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

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

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

application's message queue as a signal for the application to terminate when the message is read from the queue.

Display and movement functions

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

Function	Description
ArrangeIconicWindows	Arranges minimized (iconic) child windows.
BeginDeferWindowPos	Initializes memory used by the **DeferWindowPos** function.
BringWindowToTop	Brings a window to the top of a stack of overlapped windows.
CloseWindow	Hides the specified window or minimizes it.
DeferWindowPos	Records positioning information for a window to be moved or resized by the **EndDeferWindowPos** function.
EndDeferWindowPos	Positions or sizes several windows simultaneously based on information recorded by the **DeferWindowPos** function.
GetClientRect	Copies the coordinates of a window's client area.
GetWindowRect	Copies the dimensions of an entire window.
GetWindowText	Copies a window caption into a buffer.
GetWindowTextLength	Returns the length (in characters) of the given window's caption or text.
IsIconic	Specifies whether a window is open or closed (iconic).
IsWindowVisible	Determines whether the given window is visible.
IsZoomed	Determines whether a window is maximized.
MoveWindow	Changes the size and position of a window.
OpenIcon	Opens the specified window.
SetWindowPos	Changes the size, position, and ordering of child or pop-up windows.
SetWindowText	Sets the window caption or text.
ShowOwnedPopups	Shows or hides all pop-up windows.
ShowWindow	Displays or removes the given window.

Input functions

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

Function	Description
EnableWindow	Enables and disables mouse and keyboard input throughout the application.
GetActiveWindow	Returns a handle to the active window.
GetCapture	Returns a handle to the window with the mouse capture.
GetCurrentTime	Retrieves the current Windows time.
GetDoubleClickTime	Retrieves the current double-click time for the mouse.
GetFocus	Retrieves the handle of the window that currently owns the input focus.
GetTickCount	Returns the number of timer ticks recorded since the system was booted.
IsWindowEnabled	Determines whether the specified window is enabled for mouse and keyboard input.
KillTimer	Kills the specified timer event.
ReleaseCapture	Releases mouse input and restores normal input processing.
SetActiveWindow	Makes a window the active window.
SetCapture	Causes mouse input to be sent to a specified window.
SetDoubleClickTime	Sets the double-click time for the mouse.
SetFocus	Assigns the input focus to a specified window.
SetSysModalWindow	Makes the specified window a system modal window.
SetTimer	Creates a system-timer event.
SwapMouseButton	Reverses the meaning of left and right mouse buttons.

Hardware functions

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

Function	Description
EnableHardwareInput	Enables or disables mouse and keyboard input throughout the application.
GetAsyncKeyState	Returns interrupt-level information about the key state.
GetInputState	Returns TRUE if there is mouse or keyboard input.
GetKBCodePage	Determines which OEM/ANSI tables are loaded.
GetKeyboardState	Copies an array that contains the state of keyboard keys.
GetKeyNameText	Retrieves a string containing the name of a key from a list maintained by the keyboard driver.
GetKeyState	Retrieves the state of a virtual key.
MapVirtualKey	Accepts a virtual-key code or scan code for a key and returns the corresponding scan code, virtual-key code, or ASCII value.
OemKeyScan	Maps OEM ASCII codes 0 through 0x0FF into the OEM scan codes and shift states.
SetKeyboardState	Sets the state of keyboard keys by altering values in an array.
VkKeyScan	Translates an ANSI character to the corresponding virtual-key code and shift state for the current keyboard.

Painting functions

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

Function	Description
BeginPaint	Prepares a window for painting.
DrawFocusRect	Draws a rectangle in the style used to indicate focus.
DrawIcon	Draws an icon.
DrawText	Draws characters of a specified string.
EndPaint	Marks the end of window repainting.
ExcludeUpdateRgn	Prevents drawing within invalid areas of a window.
FillRect	Fills a given rectangle by using the specified brush.
FrameRect	Draws a border for the given rectangle.

GetDC	Retrieves the display context for the client area.
GetUpdateRect	Copies the dimensions of a window region's bounding rectangle.
GetUpdateRgn	Copies a window's update region.
GetWindowDC	Retrieves the display context for an entire window.
GrayString	Writes the characters of a string using gray text.
InvalidateRect	Marks a rectangle for repainting.
InvalidateRgn	Marks a region for repainting.
InvertRect	Inverts the display bits of the specified rectangle.
ReleaseDC	Releases a display context.
UpdateWindow	Notifies the application when parts of a window need redrawing.
ValidateRect	Releases the specified rectangle from repainting.
ValidateRgn	Releases the specified region from repainting.

How Windows manages the display

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

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

Display context types

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

Common display context

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

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

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

Table 1.4
Defaults for a display context

Attribute	Default
Background color	White
Background mode	OPAQUE
Bitmap	No default.
Brush	WHITE_BRUSH
Brush origin	(0,0)
Clipping region	Entire client area with the update region clipped as appropriate. Child and pop-up windows in the client area may also be clipped.
Color palette	DEFAULT_PALETTE
Current pen position	(0,0)

Table 1.4: Defaults for a display context (continued)

Device origin	Upper-left corner of client area.
Drawing mode	R2_COPYPEN
Font	SYSTEM_FONT (SYSTEM_FIXED_FONT for applications written to run with Windows versions prior to 3.0)
Intercharacter spacing	0
Mapping mode	MM_TEXT
Pen	BLACK_PEN
Polygon-filling mode	ALTERNATE
Relative-absolute flag	ABSOLUTE
Stretching mode	BLACKONWHITE
Text color	Black
Viewport extent	(1,1)
Viewport origin	(0,0)
Window extents	(1,1)
Window origin	(0,0)

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

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

Class display context

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

A class display context must be retrieved before it can be used, but it does not have to be released after use. As long as only one window from the class uses the context, the class display context can be kept and reused. If another window in the class needs to use the context, that window must retrieve it before any drawing occurs. Retrieving the context sets the correct origin and clipping for the new window and ensures that the context will be applied to the correct window. A handle to the class display context can be retrieved by using the **GetDC** or **BeginPaint** function. The

ReleaseDC and **EndPaint** functions have no effect on the class display context.

A class display context is given the same default selections as a common display context when the first window of the class is created (see Table 1.4, on page 46). These selections can be modified at any time. Windows preserves all new selections made for the class display context, except for the clipping region and device origin, which are adjusted for the current window when the context is retrieved. Otherwise, all other attributes remain unchanged. This means a change made by one window applies to all windows that subsequently use the context.

Changing the mapping mode of a class display context may have an undesirable effect on how a window's background is erased. For more information, see "Window background," page 52, and "Mapping functions," page 100.

Private display context

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

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

A private display context is given the same default selections as a common display context when the window is created (see Table 1.4, page 46). These selections can be modified at any time. Windows preserves any new selections made for the context. New selections, such as clipping region and brush, remain selected until the window specifically makes a change.

Changing the mapping mode of a private display context may have an undesirable effect on how the window's background is erased. For more information, see "Window background," on page 52, and "Mapping functions," on page 100.

Window display context

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

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

Windows always sets the current selections of a window display context to the same default selections as a common display context and does not preserve any change the window may have made to these selections (see Table 1.4, on page 46). Windows does not allow private or class window display contexts, so CS_OWNDC and CS_CLASSDC class styles have no effect on the window display context.

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

Display-context cache

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

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

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

Painting sequence

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

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

WM_PAINT message

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

The **InvalidateRect** and **InvalidateRgn** functions do not actually generate WM_PAINT messages. Instead, Windows accumulates the changes made by these functions and its own changes while a

window processes other messages in its application queue. Postponing the WM_PAINT message lets a window process all changes at once instead of updating bits and pieces in time-consuming individual steps.

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

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

For more information about the clipping region, see "Clipping functions," on page 106.

Windows automatically selects the update region as the clipping region of the display context. Since GDI discards (clips) drawing that extends outside the clipping region, only drawing that is in the update region is actually visible.

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

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

Update region

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

The **InvalidateRect** and **InvalidateRgn** functions add a given rectangle or region to the update region. The rectangle or region must be given in client coordinates. The update region itself is defined in client coordinates. Windows adds its own rectangles

and regions to a window's update region after operations such as moving, sizing, and scrolling the window.

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

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

Window background

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

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

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

Brush alignment

Brush alignment is particularly important on the system display where scrolling and moving are commonplace. A brush is a pattern of bits with a minimum size of 8-by-8 bits. GDI paints with a brush by repeating the pattern again and again within a given rectangle or region. If the region is moved by an arbitrary amount—for example, if the window is scrolled—and the brush is used again to filled empty areas around the original area, there is no guarantee that the original pattern and the new pattern will be

aligned. For example, if the scroll moves the original filled area up one pixel, the intersection of the original area and any new painting will be out of alignment by one pixel, or bit. Depending on the pattern, this may have a undesirable visual effect.

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

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

Painting rectangular areas

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

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

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

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

Drawing icons

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

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

memory. An application can free memory used to store an icon created by **CreateIcon** by calling **DeleteIcon**.

Drawing formatted text

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

Table 1.5
Drawing format styles

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

Table 1.5: Drawing format styles (continued)

DT_VCENTER	Vertically centered (single line only).
DT_WORDBREAK	Sets word breaks. Lines are automatically broken between words if a word would extend past the edge of the rectangle specified by the *lpRect* parameter. Carriage-return/linefeed sequence also causes a line break. Word-break characters are space, tab, carriage return, linefeed, and carriage-return/linefeed combinations. Applies to multiple-line formatting only.

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

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

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

Table 1.6
Control characters and actions

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

Drawing gray text

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

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

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

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

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

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

The application-supplied function is also defined as a parameter. If a non-NULL value is given for the function, **GrayString** automatically calls the application-supplied function instead of the **TextOut** function and passes it a handle to the display context

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

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

Nonclient-area painting

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

Dialog box functions

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

Function	Description
CheckDlgButton	Places/removes a check, or changes the state of the three-state button.
CheckRadioButton	Checks a specified button and removes checks from all others.
CreateDialog	Creates a modeless dialog box.

CreateDialogIndirect	Creates a modeless dialog box from a template.
CreateDialogIndirectParam	Creates a modeless dialog box from a template and passes data to it when it is created.
CreateDialogParam	Creates a modeless dialog box and passes data to it when it is created.
DefDlgProc	Provides default processing for any Windows messages that a dialog box with a private window class does not process.
DialogBox	Creates a modal dialog box.
DialogBoxIndirect	Creates a modal dialog box from a template.
DialogBoxIndirectParam	Creates a modal dialog box from a template and passes data to it when it is created.
DialogBoxParam	Creates a modal dialog box and passes data to it when it is created.
DlgDirList	Fills the list box with names of files matching a path.
DlgDirListComboBox	Fills a combo box with names of files matching a path.
DlgDirSelect	Copies the current selection from a list box to a string.
DlgDirSelectComboBox	Copies the current selection from a combo box to a string.
EndDialog	Frees resources and destroys windows associated with a modal dialog box.
GetDialogBaseUnits	Retrieves the base dialog units used by Windows when creating a dialog box.
GetDlgCtrlID	Returns the ID value of a control window.
GetDlgItem	Retrieves the handle of a dialog item from the given dialog box.
GetDlgItemInt	Translates the control text of an item into an integer value.
GetDlgItemText	Copies an item's control text into a string.
GetNextDlgGroupItem	Returns the window handle of the next item in a group.
GetNextDlgTabItem	Returns the window handle of the next or previous item.
IsDialogMessage	Determines whether a message is intended for the given dialog box.
IsDlgButtonChecked	Tests whether a button is checked.
MapDialogRect	Converts the dialog-box coordinates to client coordinates.
SendDlgItemMessage	Sends a message to an item within a dialog box.
SetDlgItemInt	Sets the caption or text of an item to a string that represents an integer.

SetDlgItemText	Sets the caption or text of an item to a string.

Uses for dialog boxes

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

Modeless dialog box

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

An application with a modeless dialog box processes messages for that box by using the **IsDialogMessage** function inside the main message loop.

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

Modal dialog box

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

some other operation unless the command is canceled or additional information is provided.

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

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

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

System-modal dialog box

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

Creating a dialog box

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

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

Dialog box template

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

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

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

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

Dialog box measurements

Dialog box and control dimensions and coordinates are device independent. Since a dialog box may be displayed on system displays that have widely varying pixel resolutions, dialog-box dimensions are specified in system character widths and heights instead of pixels. Characters are guaranteed to give the best possible appearance for a given display. One unit in the x direction is equal to 1/4 of the dialog base width unit. One unit in the y direction is equal to 1/8 of the dialog base height unit. The dialog base units are computed from the height and width of the system font; the **GetDialogBaseUnits** function returns the dialog base units for the current display. Applications can convert these measurements to pixels by using the **MapDialogRect** function.

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

Return values from a dialog box

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

Modeless dialog boxes cannot return values in this way since they do not use the **EndDialog** function to terminate execution and do not return control in the same way a modal dialog box does. Instead, modeless dialog boxes return values to their parent windows by using the **SendMessage** function to send a notification message to the parent window. Although Windows

does not explicitly define the content of a notification message, most applications use a WM_COMMAND message with an integer value that identifies the dialog box in the *wParam* parameter and the return value in the *lParam* parameter. Modal dialog boxes may also use this technique to return values to their parent windows before terminating.

Controls in a dialog box

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

Control identifiers

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

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

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

General control styles

The WS_TABSTOP style specifies that the user can move the input focus to the given control by pressing the TAB or SHIFT+TAB keys. Typically, every control in the dialog box has this style, so the user can move the input focus from one control to the other. If two or more controls are in the dialog box, the TAB key moves the input focus to the controls in the order in which they have been

created. The SHIFT+TAB keys move the input focus in reverse order. For modal dialog boxes, the TAB and SHIFT+TAB keys are automatically enabled for moving the input focus. For modeless dialog boxes, the **IsDialogMessage** function must be used to filter messages for the dialog box and to process these key strokes. Otherwise, the keys have no special meaning and the WS_TABSTOP style is ignored.

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

Buttons

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

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

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

function can use the **CheckRadioButton** function to set the button and automatically clear any other radio button.

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

Edit controls

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

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

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

List boxes and directory listings

Some dialog boxes display lists, such as filenames, from which the user can select one or more names. Dialog boxes that display a list typically use list-box controls. Dialog boxes that display a list of filenames typically use a list-box

control and the **DlgDirList** and **DlgDirSelect** functions. The **DlgDirList** function automatically fills a list box with the filenames in the current directory. The **DlgDirSelect** function retrieves the selected filename from the list box. Together they provide a convenient way for a dialog box to display a directory listing, and

let the user select a file without having to type in the name of the directory and file.

Combo boxes

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

Owner-draw dialog box controls

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

Table 1.7
Dialog box controls

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

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

When Windows first creates a dialog box containing owner-draw controls, it sends the owner a WM_MEASUREITEM message for each owner-draw control. The *lParam* parameter of this message contains a pointer to a **MEASUREITEMSTRUCT** data structure. When the owner receives the message for a control, the owner fills in the appropriate fields of the structure and returns. This informs Windows of the dimensions of the control or of its items so that

Windows can appropriately detect the user's interaction with the control. If a list box or combo box is created with the LBS_OWNERDRAWVARIABLE or CBS_OWNERDRAWVARIABLE style, this message is sent to the owner for each item in the control, since each item can differ in height. Otherwise, this message is sent once for the entire owner-draw control.

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

Messages for dialog box controls

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

Dialog box keyboard interface

Windows provides a special keyboard interface for modal dialog boxes and modeless dialog boxes that use the **IsDialogMessage** function to filter messages. This keyboard interface carries out special processing for several keys and generates messages that correspond to certain buttons in the dialog box or changes the input focus from one control to another. Table 1.8 lists the keys used in this interface and the respective action:

Table 1.8
Dialog box keyboard interface

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

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

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

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

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

Scrolling in dialog boxes

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

Scrolling functions

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

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

Standard scroll bars and scroll-bar controls

A standard scroll bar is a part of the nonclient area of a window. It is created with the window and displayed when the window is displayed. The sole purpose of a standard scroll bar is to let users generate scrolling requests for the window's client area. A

window has standard scroll bars if it is created with the WS_VSCROLL or WS_HSCROLL style. A standard scroll bar is either vertical or horizontal. A vertical bar always appears at the right of the client area; a horizontal bar always appears at the bottom. A standard scroll bar always has the standard scroll-bar height and width as defined by the SM_CXVSCROLL and SM_CYHSCROLL system metric values.

*(For more information, see the **GetSystemMetrics** function in Chapter 4, "Functions directory.")*

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

Scroll-bar thumb

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

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

A thumb position is an integer. The position is relative to the left or upper end of the scroll bar, depending on whether the scroll bar is horizontal or vertical. The position must be within the scroll-bar range, which is defined by minimum and maximum values. The positions are distributed equally along the scroll bar. For example, if the range is 0 to 100, there are 100 positions along the scroll bar, each equally spaced so that position 50 is in the middle of the scroll bar. The initial range depends on the scroll bar. Standard scroll bars have an initial range of 0 to 100; scroll-bar controls have an empty range (both minimum and maximum

values are zero) if no explicit range is given when the control is created. The **SetScrollRange** function sets new minimum and maximum values so that applications can change the range at any time. The **GetScrollRange** function retrieves the current minimum and maximum values. The minimum and maximum values can be any integers. For example, a spreadsheet program with 255 rows can set the vertical scroll range to 1 to 255.

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

Scrolling requests

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

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

Processing scroll messages

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

example, a word-processing application can scroll to the next line, the next page, or to the end of the document.

Scrolling the client area

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

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

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

Hiding a standard scroll bar

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

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

Menu functions

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

Function	Description
AppendMenu	Appends a menu item to a menu.
CheckMenuItem	Places or removes checkmarks next to pop-up menu items.
CreateMenu	Creates an empty menu.
CreatePopupMenu	Creates an empty pop-up menu.
DeleteMenu	Removes a menu item and destroys any associated pop-up menus.
DestroyMenu	Destroys the specified menu.
DrawMenuBar	Redraws a menu bar.
EnableMenuItem	Enables, disables, or grays a menu item.
GetMenu	Retrieves a handle to the menu of a specified window.
GetMenuCheckMarkDimensions	Retrieves the dimensions of the default menu checkmark bitmap.
GetMenuItemCount	Returns the count of items in a menu.
GetMenuItemID	Returns the item's identification.
GetMenuState	Obtains the status of a menu item.
GetMenuString	Copies a menu label into a string.
GetSubMenu	Retrieves the menu handle of a pop-up menu.
GetSystemMenu	Accesses the System menu for copying and modification.
HiliteMenuItem	Highlights or removes the highlighting from a top-level (menu-bar) menu item.
InsertMenu	Inserts a menu item in a menu.
LoadMenuIndirect	Loads a menu resource.
ModifyMenu	Changes a menu item.
RemoveMenu	Removes an item from a menu but does not destroy it.
SetMenu	Specifies a new menu for a window.
SetMenuItemBitmaps	Associates bitmaps with a menu item for display when an item is and is not checked.
TrackPopupMenu	Displays a pop-up menu at a specified screen location and tracks user interaction with the menu.

Information functions

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

Function	Description
AnyPopup	Indicates whether any pop-up window exists.
ChildWindowFromPoint	Determines which child window contains a specific point.
EnumChildWindows	Enumerates the child windows that belong to a specific parent window.
EnumTaskWindows	Enumerates all windows associated with a given task.
EnumWindows	Enumerates windows on the display.
FindWindow	Returns the handle of a window with the given class and caption.
GetNextWindow	Returns a handle to the next or previous window.
GetParent	Retrieves the handle of the specified window's parent window.
GetTopWindow	Returns a handle to the top-level child window.
GetWindow	Returns a handle from the window manager's list.
GetWindowTask	Returns the handle of a task associated with a window.
IsChild	Determines whether a window is the descendent of a specified window.
IsWindow	Determines whether a window is a valid, existing window.
SetParent	Changes the parent window of a child window.
WindowFromPoint	Identifies the window containing a specified point.

System functions

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

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

Clipboard functions

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

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

Error functions

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

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

Caret functions

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

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

Creating and displaying a caret

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

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

Figure 1.1
Caret shapes

Underline

Vertical line|

Solid block

Gray block

Bitmap

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

Sharing the caret

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

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

Cursor functions

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

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

Pointing devices and the cursor

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

Displaying and hiding the cursor

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

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

Positioning the cursor

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

The cursor hotspot and confining the cursor

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

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

Creating a custom cursor

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

Hook functions

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

Function	Description
CallMsgFilter	Passes a message and other data to the current message-filter function.
DefHookProc	Calls the next filter function in a filter-function chain.
SetWindowsHook	Installs a system and/or application filter function.
UnhookWindowsHook	Removes a Windows filter function from a filter-function chain.

Filter-function chain

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

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

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

There are five types of standard window hooks and two types of debugging hooks. The following table lists each type and describes its purpose:

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

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

Installing a filter function

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

Export the function in its module definition file.

Obtain the function's address by using the **MakeProcInstance** function.

Call the **SetWindowsHook** function, specifying the type of hook function and the address of the function (returned by **MakeProcInstance**).

Store the return value from **SetWindowsHook** in a reserved location. This value is the address of the previous filter function.

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

Property functions

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

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

Using property lists

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

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

Figure 1.2
Property list

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

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

For more information, see "Clipboard functions," on page 74.

Although the purpose of the property list is to associate data with a window for use by the application that owns the window, the handles in a property list are actually accessible to any application that has access to the window. This means an application can retrieve and use a data handle from the property list of a window created by another application. But using another application's data handles must be done with care. Only shared, global memory objects, such as GDI drawing objects, can be used by other applications. If a property list contains local or global memory handles or resource handles, only the application that

has created the window may use them. Global memory handles can be shared with other applications by using the Windows clipboard. Local memory handles cannot be shared.

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

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

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

Rectangle functions

Rectangle functions alter and obtain information about rectangles in a window's client area. In Windows, a rectangle is defined by a **RECT** data structure. The structure contains two points: the upper-left and lower-right corners of the rectangle. The sides of a

rectangle extend from these two points and are parallel to the x- and y-axes. The following list briefly describes each rectangle function:

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

Using rectangles in a Windows application

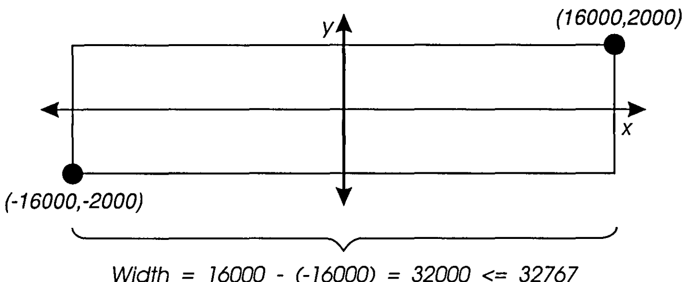
Rectangles are used to specify rectangular areas on the display or in a window, such as the cursor clipping area, the client repaint area, a formatting area for formatted text, and the scroll area. Rectangles are also used to fill, frame, or invert an area in the client area with a given brush, and to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

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

Rectangle coordinates

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

Figure 1.3
Rectangle limits



Creating and manipulating rectangles

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

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

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

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

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

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

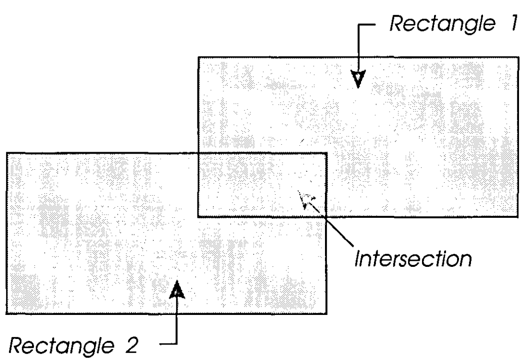


Figure 1.4
Intersection of two rectangles

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

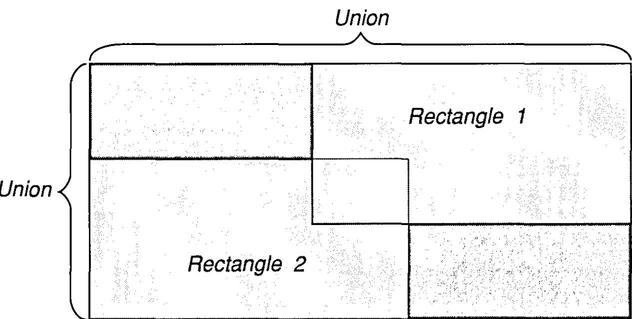


Figure 1.5
Union of two rectangles

For information about functions that draw ellipses and polygons, see "Ellipse and polygon functions," on page 109. For more information on topics related to window manager interface functions, see the following:

Topic	Reference
Function descriptions	*Reference, Volume 1*: Chapter 4, "Functions directory"
Windows messages	*Reference, Volume 1*: Chapter 5, "Messages overview," and Chapter 6, "Messages directory"
Windows data types and structures	*Reference, Volume 2*: Chapter 7, "Data types and structures"
Using the Resource Compiler	*Reference, Volume 2*: Chapter 8, "Resource script statements"

C H A P T E R

2

Graphics device interface functions

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

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

Device-context functions

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

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

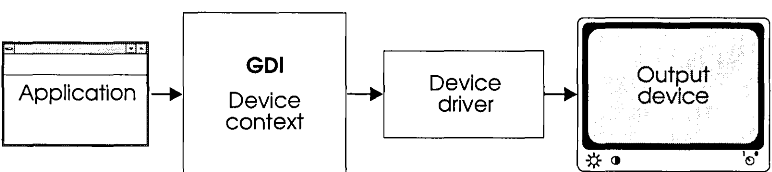


Figure 2.1
Information flow to an output device

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

The following list briefly describes each device-context function:

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

Device-context attributes

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

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

Table 2.1
Default device-context attributes and related GDI functions

Attribute	Default	GDI Functions
Background color	White	**SetBkColor**
Background mode	OPAQUE	**SetBkMode**
Bitmap	No default	**CreateBitmap** **CreateBitmapIndirect** **CreateCompatibleBitmap** **SelectObject**
Brush	WHITE_BRUSH	**CreateBrushIndirect** **CreateDIBPatternBrush** **CreateHatchBrush** **CreatePatternBrush** **CreateSolidBrush** **SelectObject**
Brush origin	(0,0)	**SetBrushOrg** **UnrealizeObject**
Clipping region	Display surface	**ExcludeClipRect** **IntersectClipRect** **OffsetClipRgn** **SelectClipRgn**
Color palette	DEFAULT_PALETTE	**CreatePalette** **RealizePalette** **SelectPalette**
Current pen position	(0,0)	**MoveTo**
Drawing mode	R2_COPYPEN	**SetROP2**
Font	SYSTEM_FONT	**CreateFont** **CreateFontIndirect** **SelectObject**
Intercharacter spacing	0	**SetTextCharacterExtra**
Mapping mode	MM_TEXT	**SetMapMode**
Pen	BLACK_PEN	**CreatePen** **CreatePenIndirect** **SelectObject**
Polygon-filling mode	ALTERNATE	**SetPolyFillMode**
Stretching mode	BLACKONWHITE	**SetStretchBltMode**
Text color	Black	**SetTextColor**
Viewport extent	(1,1)	**SetViewportExt**
Viewport origin	(0,0)	**SetViewportOrg**
Window extent	(1,1)	**SetWindowExt**
Window origin	(0,0)	**SetWindowOrg**

Saving a device context

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

Deleting a device context

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

Compatible device contexts

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

Information contexts

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

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

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

Drawing-tool functions

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

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

Drawing-tool uses

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

Brushes

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

- Black
- Dark-Gray
- Gray
- Hollow
- Light-Gray
- Null
- White

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

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

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

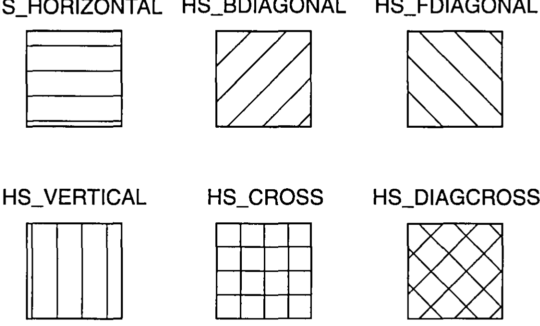


Figure 2.2
Hatched brush patterns

Pens

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

- Black
- Null
- White

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

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

Figure 2.3 Pen patterns

Color

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

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

"Color palette functions," on page 95 describes Windows color palettes and the functions used by an application to exploit their capabilities.

The second and third methods require the application to create a logical palette.

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

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

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

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

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

Color-palette functions

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

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

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

Function	Description
AnimatePalette	Replaces entries in a logical palette; Windows maps the new entries into the system palette immediately.
CreatePalette	Creates a logical palette.
GetNearestPaletteIndex	Retrieves the index of a logical palette entry most nearly matching a specified RGB value.
GetPaletteEntries	Retrieves entries from a logical palette.
GetSystemPaletteEntries	Retrieves a range of palette entries from the system palette.

GetSystemPaletteUse	Determines whether an application has access to the full system palette.
RealizePalette	Maps entries in a logical palette to the system palette.
SelectPalette	Selects a logical palette into a device context.
SetPaletteEntries	Sets new palette entries in a logical palette; Windows does not map the new entries to the system palette until the application realizes the logical palette.
SetSystemPaletteUse	Allows an application to use the full system palette.
UpdateColors	Performs a pixel-by-pixel translation of each pixel's current color to the system palette. This allows an inactive window to correct its colors without redrawing its client area.

How color palettes work

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

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

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

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

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

Finally, when all entries in the system palette have been used, Windows takes these logical palette entries that do not exactly match and matches them as closely as possible to entries already

in the system palette. To further aid this color matching, Windows sets aside 20 static colors (called the "default palette") in the system palette to which it can match entries in a background palette.

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

Figure 2.4
Palette manager color-mapping algorithm

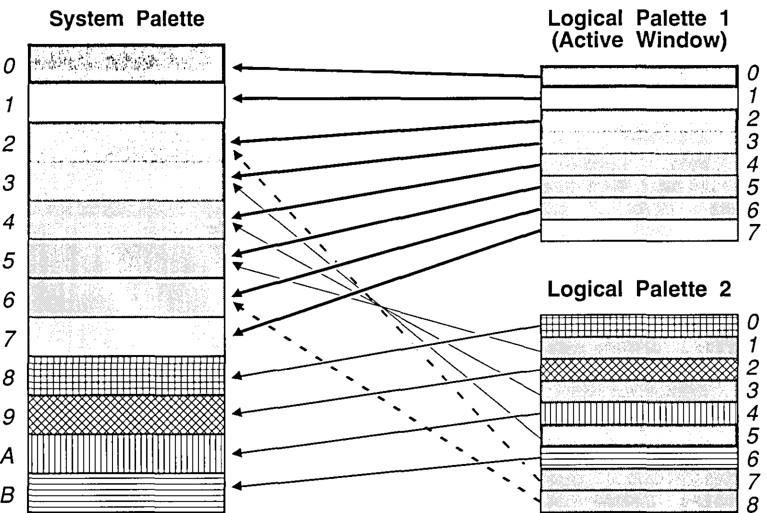


Figure 2.4 illustrates this process. In this figure, a hypothetical display has a system palette capable of containing 12 colors. The application that created Logical Palette 1 owns the active window and was the first to realize its logical palette, which consists of 8 colors. Logical Palette 2 is owned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

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

Three of the colors (1, 3, and 5) in Logical Palette 2 are identical to colors in the system palette; to save space in the palette, then, Windows simply matched those colors to the existing system colors when the second application realized its palette. Colors 0, 2,

4, and 6 were not already in the system palette, however, and so Windows mapped those colors into the system palette.

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

Using a color palette

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

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

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

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

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

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

Drawing-attribute functions

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

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

Background mode and color

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

state. The **SetBkMode** and **SetBkColor** functions accomplish this task.

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

Stretch mode

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

Text color

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

Mapping functions

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

Function	Description
GetMapMode	Retrieves the current mapping mode.
GetViewportExt	Retrieves a device context's viewport extents.
GetViewportOrg	Retrieves a device context's viewport origin.

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

There are eight different mapping modes: MM_ANISOTROPIC, MM_HIENGLISH, MM_HIMETRIC, MM_ISOTROPIC, MM_LOENGLISH, MM_LOMETRIC, MM_TEXT, and MM_TWIPS. Each mode has a specific use in a Windows application. Table 2.1 summarizes the eight GDI mapping modes:

GDI mapping modes

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

Constrained mapping modes

GDI classifies six of the mapping modes as constrained mapping modes: MM_HIENGLISH, MM_HIMETRIC, MM_LOENGLISH, MM_LOMETRIC, MM_TEXT, and MM_TWIPS. In each of these modes, one logical unit is mapped to a predefined physical unit. For instance, the MM_TEXT mode maps one logical unit to one device pixel, and the MM_LOENGLISH mode maps one logical unit to 0.01 inch on the device. These mapping modes are constrained because the scaling factor is fixed, so an application cannot change the number of logical units that Windows maps to a physical unit. Table 2.1 shows the number of logical units in various mapping modes that result in a certain physical unit:

Logical/physical conversion table

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

Partially constrained and unconstrained mapping modes

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

Partially constrained mapping mode

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

Unconstrained mapping mode

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

Transformation equations

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

- Transforming logical points to device points:

  Dx = (Lx – xWO) * xVE/xWE + xVO

  Dy = (Ly – yWO) * yVE/yWE + yVO
- Transforming device points to logical points:

  Lx = (Dx – xVO) * xWE/xVE + xWO

  Ly = (Dy – yVO) * yWE/yVE + yWO

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

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

Ly	Logical-coordinate system *y*-coordinate
Dx	Device *x*-coordinate
Dy	Device *y*-coordinate

The following four ratios are scaling factors:

xVE/xWE
yVE/yWE
xWE/xVE
yWE/yVE

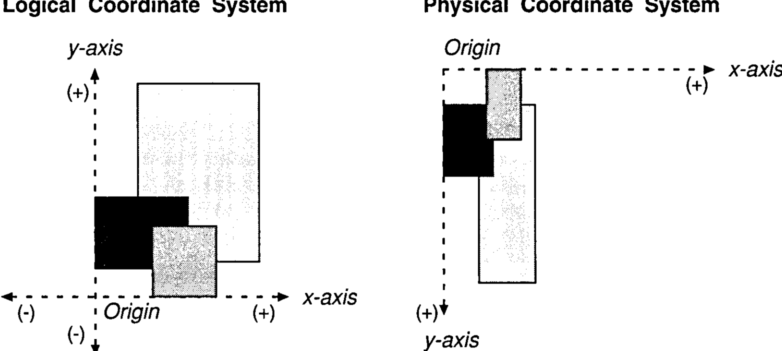
They are used to determine the necessary stretching or compressing of logical units. The subtraction and addition of viewport and window origins is referred to as the translational component of the equation.

Example: MM_TEXT

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

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

Figure 2.5
Mapping with MM_TEXT



Example: MM_LOENGLISH

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

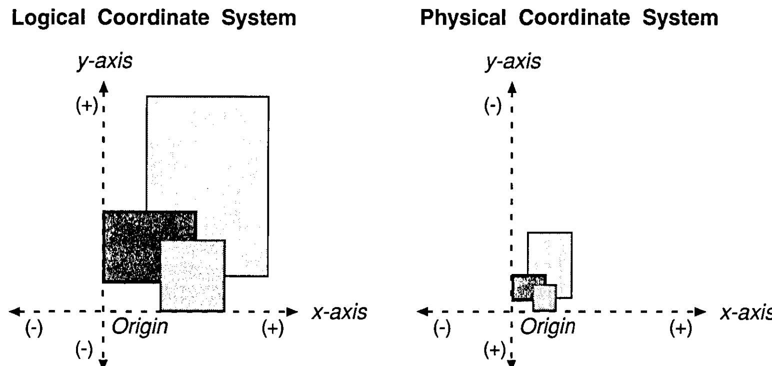


Figure 2.6 Mapping with MM_LOENGLISH

Coordinate functions

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

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

ScreenToClient	Converts screen coordinates into client coordinates.
WindowFromPoint	Determines which window contains a specified point.

Region functions

For more information about clipping functions, see "Clipping functions" on page 106.

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

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

Clipping functions

Clipping functions create, test, and alter clipping regions. A clipping region is the portion of a window's client area where GDI

creates output; any output sent to that portion of the client area which is outside the clipping region will not be visible. Clipping regions are useful in any Windows application that needs to save one part of the client area and simultaneously send output to another. The following list briefly describes each clipping function:

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

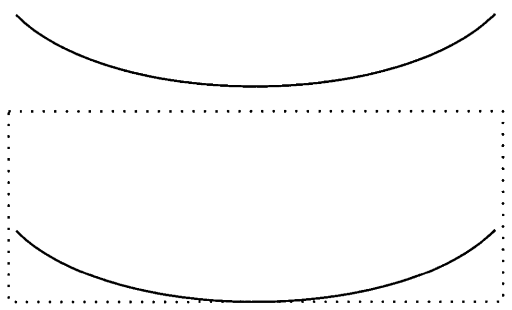
Line-output functions

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

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

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

Figure 2.7
Arc and its bounding rectangle



Function coordinates

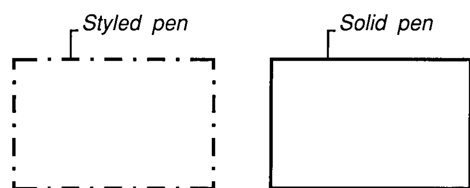
Line-output functions require coordinates in logical units, which GDI uses to draw a line in logical space. The use of logical units ensures device independence in Windows. GDI maps this line from the logical space to the physical space on the device. The number of logical units that GDI maps to a pixel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

Pen styles, colors, widths

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

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

Figure 2.8
Styled-Pen and Solid-Pen Rectangles



Ellipse and polygon functions

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

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

Function coordinates

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

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

specified by ($X2$, $Y2$), the rectangle's upper-left corner will be ($X1$, $Y1$) and the lower-right corner will be ($X2 - 1$, $Y2 - 1$).

Bounding rectangles

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

For information about functions that alter or obtain information about rectangles in a window's client area, see "Rectangle functions," on page 82.

Bitmap functions

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

Function	Description
BitBlt	Copies a bitmap from a source to a destination device.
CreateBitmap	Creates a bitmap.
CreateBitmapIndirect	Creates a bitmap described in a data structure.
CreateCompatibleBitmap	Creates a bitmap that is compatible with a specified device.
CreateDiscardableBitmap	Creates a discardable bitmap that is compatible with a specified device.
ExtFloodFill	Fills the display surface within a border or over an area of a given color.
FloodFill	Fills the display surface within a border.
GetBitmapBits	Retrieves the bits in memory for a specific bitmap.
GetBitmapDimension	Retrieves the dimensions of a bitmap.
GetPixel	Retrieves the RGB value for a pixel.
LoadBitmap	Loads a bitmap from a resource file.
PatBlt	Creates a bit pattern.
SetBitmapBits	Sets the bits of a bitmap.

SetBitmapDimension	Sets the height and width of a bitmap.
SetPixel	Sets the RGB value for a pixel.
StretchBlt	Copies a bitmap from a source to a destination device (compresses or stretches, if necessary).

Bitmaps and devices

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

Device-independent bitmap functions

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

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

A device-independent bitmap specification consists of two parts:

1. A **BITMAPINFO** data structure that defines the format of the bitmap and optionally supplies a table of colors used by the bitmap

2. An array of bytes that contain the bitmap bit values

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

Text functions

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

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

Font functions

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

The following list briefly describes each font function:

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

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

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

The Helvetica typeface is an example of a familiar typeface. It belongs to the Swiss font family. Available fonts within this typeface include 8-point Helvetica bold and 10-point Helvetica italic.

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

Figure 2.9
Fonts from two typefaces

This is a line of 12 point Helvetica.

This is a line of 12 point Helvetica bold.

This is a line of 12 point Helvetica italic.

This is a line of 12 point Courier.

This is a line of 12 point Courier bold.

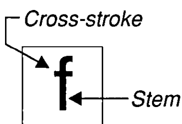
This is a line of 12 point Courier italic.

Font family

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

Figure 2.10 shows a lowercase *f* composed of a cross-stroke and a stem with a loop at the top:

Figure 2.10
Cross-stroke and stem



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

Figure 2.11
Serifs

GDI uses five distinct family names to categorize typefaces and fonts. A sixth name is used for generic cases. Note that GDI's family names do not correspond to traditional typographic categories. Table 2.1 lists the font-family names and briefly describes each family:

Font families

Name	Description
Dontcare	Generic family name. Used when information about a font does not exist or does not matter.
Decorative	Novelty fonts. Old English, for example.
Modern	Constant stroke width (fixed-pitch), with or without serifs. Fixed-pitch fonts are usually modern. Pica, Elite, and Courier, for example.
Roman	Variable stroke width (proportionally spaced), with serifs. Times Roman, Palatino, and Century Schoolbook, for example.

Script	Designed to look like handwriting. Script and Cursive, for example.
Swiss	Variable stroke width (proportionally spaced), without serifs. Helvetica and Swiss, for example.

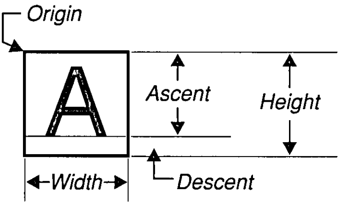
Character cells

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

Measurement	Description
Ascent	Specifies the distance in character-cell rows from the character-cell baseline to the top of the character cell.
Baseline	Serves as the base on which all characters stand (some lowercase letters have descenders, such as the tail of the *g* or *y*, that descend below the baseline).
Descent	Specifies the distance in character-cell rows from the character-cell baseline to the bottom of the character cell.
Height	Specifies the height of a character-cell row.
Origin	Used as a point of reference when the character is written on a device or a display surface. The origin is the upper-left corner of the character cell.
Width	Specifies the width of a character-cell column.

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

Figure 2.12
Character-cell dimensions



Altering characters

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

Italic

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

Bold

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

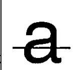
Underline

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

Strikeout

A strikeout font has a solid horizontal line drawn through each character. The position of this line within each character cell is constant for a given font. Figure 2.13 shows characters that are struck out:

Figure 2.13
Strikeout characters



This string of text illustrates ~~the effect of implementing the strikeout attribute.~~

Leading

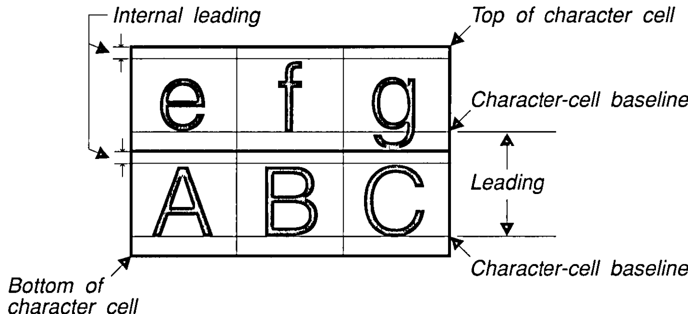
Leading is the distance from baseline to baseline of two adjacent rows of text. When font designers develop a font, they specify that a given amount of space should appear between rows. The addition of this space ensures that a character is not obscured by part of another character in an adjacent row.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

between rows of text as they are printed on a device (external leading).

Internal leading

Internal leading refers to the space inserted within character cells of a particular font. Only marks such as accents, umlauts, and tildes in foreign character sets appear within the space allocated for internal leading. Figure 2.14 shows two rows of text that use internal leading:

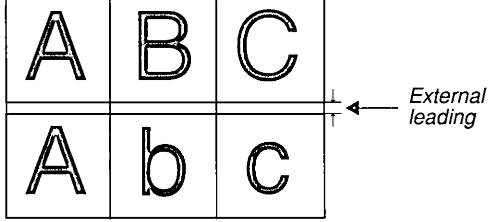
Figure 2.14
Internal leading



External leading

External leading is space inserted between the top and bottom of character cells in adjacent rows of text. The font designer must specify the amount of external leading necessary to produce easily readable text from a particular font. External leading is not built into a font; you must add it before you print text on a device. Figure 2.15 shows external leading:

Figure 2.15
External leading



Character set

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

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

- ANSI
- OEM
- Symbol
- Vendor specific

ANSI character set

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

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

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

OEM character set

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

Symbol character set

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

Vendor-specific character sets

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

Pitch

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

Average character width

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

Maximum character width

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

Digitized aspect

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

Overhang

When a particular font is not available on a device, GDI sometimes synthesizes that font. The process of synthesizing may add width or height to an existing font.

Whenever GDI synthesizes an italic or bold font from a normal font, extra columns are added to individual character cells in that font. The difference in width (the extra columns) between a string created with the normal font and a string created with the synthesized font is called the overhang.

Selecting fonts with GDI

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

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

The process of selecting the physical font that bears the closest resemblance to the specified logical font is known as font mapping. GDI also maintains a font table. Each entry in the font table describes a physical font and its attributes. Included in each entry is a pointer to a corresponding font resource. Figure 2.16 shows a font table that contains fonts X, Y, and Z:

Figure 2.16
A GDI font table

Font Table

Font X information				
leading	*italic*	*underline*	*weight*	
char set	*width*	*height*	*first char*	
pitch and family	*last char*	*. . .*		*Pointer to font X resource*
Font Y information				
leading	*italic*	*underline*	*weight*	
char set	*width*	*height*	*first char*	
pitch and family	*last char*	*. . .*		*Pointer to font Y resource*
Font Z information				
leading	*italic*	*underline*	*weight*	
char set	*width*	*height*	*first char*	
pitch and family	*last char*	*. . .*		*Pointer to font Z resource*

Font-mapping scheme

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

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

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

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

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

Table 2.2 lists the characteristics that are penalized by GDI's font mapper. The characteristics are grouped according to penalty weights, with the heaviest penalty assigned to the CharSet characteristic and the lightest penalty assigned to the Weight, Slant, Underline, and StrikeOut characteristics.

Table 2.2
Font-mapping characteristics

Characteristic	Penalty weight	Penalty scheme
CharSet	4	If the character set does not match, the candidate font is penalized heavily. Fonts with the wrong character set are very rarely selected as the physical font. There is no default character set. This means a logical font must alway specify the desired set.
Pitch	3	The wrong pitch is penalized heavily. If the requested pitch is fixed, a wrong pitch is assessed a greater penalty since an application that handles fixed pitches may not be able to handle variable-pitch fonts.
Family	3	If the font families do not match, the candidate font is penalized heavily. If a default font family is requested, no penalties are assessed.
FaceName	3	If the font typeface names do not match, the candidate font is penalized heavily. If a default font facename is requested, no penalties are assessed.

Table 2.2: Font-mapping characteristics (continued)

Height	2	The wrong height is penalized. GDI always chooses or synthesizes a shorter font if the exact height is not available. GDI can synthesize a font by expanding a font's character bitmaps by an integer multiple. GDI will expand a font up to eight times. If a default height is requested, GDI arbitrarily searches for a twelve-point font.
Width	2	The wrong width is penalized. GDI always chooses or synthesizes a narrower font if the exact width is not available. If a default width is requested, GDI assesses a penalty for any difference between the aspect ratio of the device and the aspect ratio of the font. The mapper can give unexpected results if there are no fonts for the given aspect ratio.
Weight	1	Although GDI can synthesize bold, an actual bold font is preferred. The mapper penalizes for synthesizing.
Slant	1	Although GDI can synthesize italics, an actual italic font is preferred. The mapper penalizes for synthesizing.
Underline	1	Although GDI can synthesize underlining, an actual underline font is preferred. The mapper penalizes for synthesizing.
StrikeOut	1	Although GDI can synthesize strikeouts, an actual strikeout font is preferred. The mapper penalizes for synthesizing.

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

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

The penalty weights listed in Table 2.2 are the default penalties used by GDI.

Example of font selection

For the purpose of this example, assume that the system font table lists only the three fonts shown in Figure 2.16, "A GDI Font Table," fonts X, Y, and Z. Suppose you need to use a specific font, font Q, to create text on an output device. You will need to describe font Q so that GDI can choose the physical font (X, Y, or Z) that bears the closest resemblance to Q.

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

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

Table 2.2 shows the physical fonts in the font table and the penalties that GDI assigns to each as it tries to find a font that will match font Q. The left column shows the font attributes that GDI compares; the second column gives the attributes of font Q, the desired font. The attributes of fonts X, Y, and Z—the fonts that are actually in the system font table—are followed by the penalty values that GDI gives to each one. The bottom row of the table gives the penalty totals for each font:

Figure 2.17
Sample font selection ratings

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

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

Font files and font resources

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

Metafile functions

Metafile functions close, copy, create, delete, retrieve, play, and return information about metafiles. A metafile is a collection of GDI commands that creates desired text or images.
Metafiles provide a convenient method of storing graphics commands that create text or images. Metafiles are especially useful in applications that use specific text or a particular image repeatedly. They are also device-independent; by creating text or images with GDI commands and then placing the commands in a metafile, an application can re-create the text or images repeatedly on a variety of devices. Metafiles are also useful in applications that need to pass graphics information to other applications.

The following list briefly describes each metafile function:

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

Creating a metafile

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

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

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

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

1. Create a special device context by using the **CreateMetaFile** function.
2. Send GDI commands to the metafile by using the special device context.
3. Close the metafile by calling the **CloseMetaFile** function. This function returns a metafile handle.

4. Display the image or text on a device by using the **PlayMetaFile** function, passing to the function the metafile handle obtained from **CloseMetaFile** and a device-context handle for the device to which the metafile is to be played.

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

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

Storing a metafile in memory or on disk

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

To store the metafile in memory, an application calls **CreateMetafile** and passes NULL as the function parameter. There are two ways of storing a metafile in a disk file:

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

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

Deleting a metafile

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

Changing how Windows plays a metafile

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

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

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

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

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

*See the description of the **HANDLETABLE** data structure in Chapter 7, "Data types and structures," in Reference, Volume 2, for info on the handle table format.*

Objects are added to the table in the order in which they are created. For example, if a brush is the first object created in a metafile, the brush is given index zero. If the second object is a pen, it is given index 1, and so on.

Printer-control functions

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

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

Printer-escape function

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

Creating output on a printer

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

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

Manager maintains this separation. The printer-escape functions affect the way Print Manager handles this separation task.

Banding output

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

Banding involves several steps:

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

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

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

NEWFRAME escape returns a message that defines the error or abort message.

Starting and ending a print job

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

Terminating a print job

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

Information escapes

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

Additional escape calls

For a detailed description of the functions that alter interword and intercharacter spacing, see Sections "Text functions," and "Font functions."

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

Environment functions

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

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

For more information on topics related to GDI functions, see the following:

Topic	Reference
Function descriptions	*Reference, Volume 1*: Chapter 4, "Functions directory"
Windows data types and structures	*Reference, Volume 2*: Chapter 7, "Data types and structures"
Metafile formats	*Reference, Volume 2*: Chapter 9, "File format"
Raster operations	*Reference, Volume 2*: Chapter 11, "Binary and ternary raster-operation codes"
Printer escapes	*Reference, Volume 2*: Chapter 12, "Printer escapes"

C H A P T E R

3

System services interface functions

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

This chapter lists the following categories of functions:

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

Module-management functions

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

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

Memory-management functions

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

Function	Description
DefineHandleTable	Creates a private handle table in an application's default data segment.

Function	Description
GetFreeSpace	Retrieves the number of bytes available in the global heap.
GetWinFlags	Retrieves information about the system memory configuration.
GlobalAlloc	Allocates memory from the global heap.
GlobalCompact	Compacts global memory to generate free bytes.
GlobalDiscard	Discards a global memory block if the lock count is zero, but does not invalidate the handle of the memory block.
GlobalDosAlloc	Allocates global memory that can be accessed by DOS running in real or protected mode.
GlobalDosFree	Frees global memory previously allocated by the **GlobalDosAlloc** function.
GlobalFlags	Returns the flags and lock count of a global memory block.
GlobalFree	Removes a global memory block and invalidates the handle of the memory block.
GlobalHandle	Retrieves the handle of a global memory object.
GlobalLock	Retrieves a pointer to a global memory block specified by a handle. Except for nondiscardable objects in protected (standard or 386 enhanced) mode, the block is locked into memory at the given address and its lock count is increased by one.
GlobalLRUNewest	Moves a global memory object to the newest least-recently-used (LRU) position.
GlobalLRUOldest	Moves a global memory object to the oldest least-recently-used (LRU) position.
GlobalNotify	Installs a notification procedure for the current task.
GlobalReAlloc	Reallocates a global memory block.
GlobalSize	Returns the size (in bytes) of a global memory block.
GlobalUnlock	Invalidates the pointer to a global memory block previously retrieved by the **GlobalLock** function. In real mode, or if the block is discardable, **GlobalUnlock** decreases the block's lock count by one.
GlobalUnwire	Decreases the lock count set by the **GlobalWire** function, and unlocks the memory block if the count is zero.
GlobalWire	Moves an object to low memory and increases the lock count.
LimitEMSPages	Limits the amount of expanded memory that Windows will assign to an application.
LocalAlloc	Allocates memory from the local heap.
LocalCompact	Compacts local memory.

LocalDiscard	Discards a local memory block if the lock count is zero, but does not invalidate the handle of the memory block.
LocalFlags	Returns the memory type of a local memory block.
LocalFree	Frees a local memory block from memory if the lock count is zero and invalidates the handle of the memory block.
LocalHandle	Retrieves the handle of a local memory object.
LocalInit	Initializes a local heap in the specified segment.
LocalLock	Locks a block of local memory by increasing its lock count.
LocalReAlloc	Reallocates a local memory block.
LocalShrink	Shrinks the local heap.
LocalSize	Returns the size (in bytes) of a local memory block.
LocalUnlock	Unlocks a local memory block.
LockData	Locks the current data segment in memory.
LockSegment	Locks a specified data segment in memory.
SetSwapAreaSize	Increases the amount of memory that an application reserves for code segments.
SwitchStackBack	Returns the stack of the current task to the task's data segment after it had been previously redirected by the **SwitchTasksBack** function.
SwitchStackTo	Changes the stack of the current task to the specified data segment, such as the data segment of a dynamic-link library (DLL).
UnlockData	Unlocks the current data segment.
UnLockSegment	Unlocks a specified data segment.

Segment functions

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

Function	Description
AllocDStoCSAlias	Accepts a data-segment selector and returns a code-segment selector that can be used to execute code in a data segment.
AllocSelector	Allocates a new selector.
ChangeSelector	Generates a temporary code selector that corresponds to a given data selector, or a

	temporary data selector that corresponds to a given code selector.
DefineHandleTable	Creates a private handle table which Windows updates automatically.
FreeSelector	Frees a selector originally allocated by the **AllocSelector** or **AllocDStoCSAlias** functions.
GetCodeInfo	Retrieves information about a code segment.
GlobalFix	Prevents a global memory block from moving in linear memory.
GlobalPageLock	Page-locks the memory associated with the specified global selector and increments its page-lock count. Memory that is page-locked cannot be moved or paged out to disk.
GlobalPageUnlock	Decrements the page-lock count for a block of memory. If the page-lock count reaches zero, the memory can be moved and paged out to disk.
GlobalUnfix	Unlocks a global memory block previously fixed by the **GlobalFix** function.
LockSegment	Locks a segment in memory.
UnlockSegment	Unlocks a segment previously locked by the **LockSegment** function.

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

Operating-system interrupt functions

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

The following list briefly describes these functions:

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

Task functions

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

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

Resource-management functions

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

Function	Description
AccessResource	Opens the specified resource.
AllocResource	Allocates uninitialized memory for a resource.
FindResource	Determines the location of a resource.
FreeResource	Removes a loaded resource from memory.
LoadAccelerators	Loads an accelerator table.
LoadBitmap	Loads a bitmap resource.
LoadCursor	Loads a cursor resource.

LoadIcon	Loads an icon resource.
LoadMenu	Loads a menu resource.
LoadResource	Loads a resource.
LoadString	Loads a string resource.
LockResource	Retrieves the absolute memory address of a resource.
SetResourceHandler	Sets up a function to load resources.
SizeofResource	Supplies the size (in bytes) of a resource.
UnlockResource	Unlocks a resource.

String-manipulation functions

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

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

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

Atom-management functions

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

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

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

Initialization-file functions

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

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

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

Communication functions

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

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

Sound functions

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

Function	Description
CloseSound	Closes the play device after flushing the voice queues and freeing the buffers.

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

Utility macros and functions

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

Function	Description
HIBYTE	Returns the high-order byte of an integer.
HIWORD	Returns the high-order word of a long integer.
LOBYTE	Returns the low-order byte of an integer.
LOWORD	Returns the low-order word of a long integer.
MAKEINTATOM	Casts an integer for use as a function argument.
MAKEINTRESOURCE	Converts an integer value into a long pointer to a string, with the high-order word of the long pointer set to zero.
MAKELONG	Creates an unsigned long integer.
MAKEPOINT	Converts a long value that contains the *x*- and *y*-coordinates of a point into a **POINT** data structure.

MulDiv	Multiplies two word-length values and then divides the result by a third word-length value, returning the result rounded to the nearest integer.
PALETTEINDEX	Converts an integer into a palette-index **COLORREF** value.
PALETTERGB	Converts three values for red, green, and blue into a palette-relative RGB **COLORREF** value.
RGB	Converts three values for red, green, and blue into an explicit RGB **COLORREF** value.

File I/O functions

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

Function	Description
GetDriveType	Determines whether a disk drive is removeable, fixed, or remote.
GetSystemDirectory	Retrieves the pathname of the Windows system subdirectory.
GetTempDrive	Returns the letter of the optimal drive for temporary file storage.
GetTempFileName	Creates a temporary filename.
GetWindowsDirectory	Retrieves the pathname of the Windows directory.
_lclose	Closes a file.
_lcreat	Creates a new file or opens and truncates an existing file.
_llseek	Positions the pointer to a file.
_lopen	Opens an existing file.
_lread	Reads data from a file.
_lwrite	Writes data in a file.
OpenFile	Creates, opens, reopens, or deletes the specified file.
SetHandleCount	Changes the number of file handles available to a task.

Debugging functions

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

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

Optimization-tool functions

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

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

Application-execution functions

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

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

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

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

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

Topic	Reference
Function descriptions	*Reference, Volume 1:* Chapter 4, "Functions directory"
Windows data types and structures	*Reference, Volume 2:* Chapter 7, "Data types and structures"
Initialization-file formats	*Reference, Volume 2:* Chapter 9, "File formats"
Diagnostic messages for debugging	*Reference, Volume 2:* Appendix C, "Windows debugging messages"

C H A P T E R

4

Functions directory

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

AccessResource

Syntax int AccessResource(hInstance, hResInfo)
function AccessResource(Instance, ResInfo: THandle): Integer;

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

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

Parameters

hInstance **HANDLE** Identifies the instance of the module whose executable file contains the resource.

hResInfo **HANDLE** Identifies the desired resource. This handle should be created by using the **FindResource** function.

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

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

AddAtom

Syntax ATOM AddAtom(lpString)
function AddAtom(Str: PChar): TAtom;

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

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

Parameters *lpString* **LPSTR** Points to the character string to be added to the table. The string must be a null-terminated character string.

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

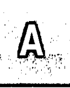
Comments The atom values returned by **AddAtom** range from 0xC000 to 0xFFFF. Atoms are case insensitive.

AddFontResource

Syntax int AddFontResource(lpFilename)
function AddFontResource(FileName: PChar): Integer;

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

Parameters *lpFilename* **LPSTR** Points to a character string that names the font-resource file or contains a handle to a loaded module. If *lpFilename* points to the font-resource filename, the string must be null-terminated, have the DOS filename format, and include the extension. If *lpFilename* contains a handle,



the handle is in the low-order word and the high-order word is zero.

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

Comments Any application that adds or removes fonts from the Windows font table should notify other windows of the change by using the **SendMessage** function with the *hWnd* parameter set to –1 to send a WM_FONTCHANGE message to all top-level windows in the system. It is good practice to remove any font resource an application has added once the application is through with the resource.

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

AdjustWindowRect

Syntax void AdjustWindowRect(lpRect, dwStyle, bMenu)
procedure AdjustWindowRect(var Rect: TRect; Style: Longint; Menu: Bool);

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

Parameters

lpRect **LPRECT** Points to a **RECT** data structure that contains the coordinates of the client rectangle.

dwStyle **DWORD** Specifies the window styles of the window whose client rectangle is to be converted.

bMenu **BOOL** Specifies whether the window has a menu.

Return value None.

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

AdjustWindowRectEx 3.0

Syntax

void AdjustWindowRectEx(lpRect, dwStyle, bMenu, dwExStyle)
procedure AdjustWindowRectEx(var Rect: TRect; Style: Longint; Menu: Bool; ExStyle: Longint);

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

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

Parameters

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

Return value None.

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

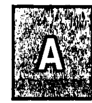
AllocDStoCSAlias 3.0

Syntax

WORD AllocDStoCSAlias(wSelector)
function AllocDStoCSAlias(Selector: Word): Word;

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

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



Parameters *wSelector* **WORD** Specifies the data-segment selector.

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

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

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

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

AllocResource

Syntax HANDLE AllocResource(hInstance, hResInfo, dwSize)
function AllocResource(Instance, ResInfo: THandle; Size: Longint): THandle;

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

Parameters

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

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

AllocSelector 3.0

Syntax WORD AllocSelector(wSelector)
function AllocSelector(Selector: Word): Word;

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

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

Parameters

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

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

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

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

AnimatePalette 3.0

Syntax void AnimatePalette(hPalette, wStartIndex, wNumEntries, lpPaletteColors)
procedure AnimatePalette(Palette: HPalette; StartIndex: Word; NumEntries: Word; var PaletteColors);

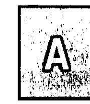
This function replaces entries in the logical palette identified by the *hPalette* parameter. When an application calls **AnimatePalette**, it does not have to update its client area because Windows maps the new entries into the system palette immediately.

Parameters

hPalette **HPALETTE** Identifies the logical palette.

wStartIndex **WORD** Specifies the first entry in the palette to be animated.

wNumEntries **WORD** Specifies the number of entries in the palette to be animated.



lpPaletteColors	**LPPALETTEENTRY** Points to the first member of an array of **PALETTEENTRY** data structures to replace the palette entries identified by *wStartIndex* and *wNumEntries*.

Return value None.

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

AnsiLower

Syntax LPSTR AnsiLower(lpString)
function AnsiLower(Str: PChar): PChar;

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

Parameters

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

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

AnsiLowerBuff 3.0

Syntax WORD AnsiLowerBuff(lpString, nLength)
function AnsiLowerBuff(Str: PChar; Length: Word): Word;

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

Parameters

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

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

AnsiNext

Syntax LPSTR AnsiNext(lpCurrentChar)
function AnsiNext(CurrentChar: PChar): PChar;

This function moves to the next character in a string.

Parameters *lpCurrentChar* **LPSTR** Points to a character in a null-terminated string.

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

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

AnsiPrev

Syntax LPSTR AnsiPrev(lpStart, lpCurrentChar)
function AnsiPrev(Start, CurrentChar: PChar): PChar;

This function moves to the previous character in a string.

Parameters *lpStart* **LPSTR** Points to the beginning of the string.

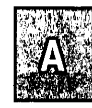
lpCurrentChar **LPSTR** Points to a character in a null-terminated string.

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

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

AnsiToOem

Syntax int AnsiToOem(lpAnsiStr, lpOemStr)
function AnsiToOem(AnsiStr, OemStr: PChar): Integer;



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

Parameters

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

Return value The return value is always –1.

AnsiToOemBuff 3.0

Syntax void AnsiToOemBuff(lpAnsiStr, lpOemStr, nLength)
procedure AnsiToOemBuff(AnsiStr, OemStr: PChar; Length: Integer);

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

Parameters

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

Return value None.

AnsiUpper

Syntax LPSTRAnsiUpper(lpString)
function AnsiUpper(Str: PChar): PChar;

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

Parameters

lpString	**LPSTR** Points to a null-terminated character string or specifies single character. If *lpString* specifies a single

character, that character is in the low-order byte of the low-order word, and the high-order word is zero.

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

AnsiUpperBuff 3.0

Syntax WORD AnsiUpperBuff(lpString, nLength)
function AnsiUpperBuff(Str:Pchar;Length:Word):Word;

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

Parameters

lpString **LPSTR** Points to a buffer containing one or more characters.

nLength **WORD** Specifies the number of characters in the buffer identified by the *lpString* parameter. If *nLength* is zero, the length is 64K (65,536).

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

AnyPopup

Syntax BOOL AnyPopup( )
function AnyPopup: Bool;

AppendMenu

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

Parameters None.

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

AppendMenu 3.0

Syntax BOOL AppendMenu(hMenu, wFlags, wIDNewItem, lpNewItem)
function AppendMenu(Menu: HMenu; Flags, IDNewItem: Word;
NewItem: PChar): Bool;

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

Parameters

hMenu — **HMENU** Identifies the menu to be changed.

wFlags — **WORD** Specifies information about the state of the new menu item when it is added to the menu. It consists of one or more values listed in the following "Comments" section.

wIDNewItem — **WORD** Specifies either the command ID of the new menu item or, if *wFlags* is set to MF_POPUP, the menu handle of the pop-up menu.

lpNewItem — **LPSTR** Specifies the content of the new menu item. The interpretation of the *lpNewItem* parameter depends upon the setting of the *wFlags* parameter.

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

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

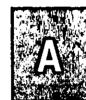
Comments Whenever a menu changes (whether or not the menu resides in a window that is displayed), the application should call **DrawMenuBar**.

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

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

The following list describes the flags that can be set in the *wFlags* parameter:

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



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

Arc

Syntax BOOL Arc(hDC, X1, Y1, X2, Y2, X3, Y3, X4, Y4)
function Arc(DC: HDC; X1, Y1, X2, Y2, X3, Y3, X4, Y4: Integer): Bool;

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

Parameters

hDC	**HDC** Identifies the device context.
X1	**int** Specifies the logical *x*-coordinate of the upper-left corner of the bounding rectangle.
Y1	**int** Specifies the logical *y*-coordinate of the upper-left corner of the bounding rectangle.
X2	**int** Specifies the logical *x*-coordinate of the lower-right corner of the bounding rectangle.
Y2	**int** Specifies the logical *y*-coordinate of the lower-right corner of the bounding rectangle.
X3	**int** Specifies the logical *x*-coordinate of the arc's starting point. This point does not have to lie exactly on the arc.
Y3	**int** Specifies the logical *y*-coordinate of the arc's starting point. This point does not have to lie exactly on the arc.

X4	**int** Specifies the logical *x*-coordinate of the arc's endpoint. This point does not have to lie exactly on the arc.
Y4	**int** Specifies the logical *y*-coordinate of the arc's endpoint. This point does not have to lie exactly on the arc.

Return value The return value specifies whether the arc is drawn. It is nonzero if the arc is drawn; otherwise, it is zero.

Comments The width of the rectangle specified by the absolute value of *X2* – *X1* must not exceed 32,767 units. This limit applies to the height of the rectangle as well.

ArrangeIconicWindows 3.0

Syntax WORD ArrangeIconicWindows(hWnd)
function ArrangeIconicWindows(Wnd: HWnd): Word;

This function arranges all the minimized (iconic) child windows of the window specified by the hWnd parameter.

Parameters

hWnd	**HWND** Identifies the window.

Return value The return value is the height of one row of icons, or zero if there were no icons.

Comments Applications that maintain their own iconic child windows call this function to arrange icons in a client window. This function also arranges icons on the desktop window, which covers the entire screen. The **GetDesktopWindow** function retrieves the window handle of the desktop window.

To arrange iconic MDI child windows in an MDI client window, an application sends the WM_MDIICONARRANGE message to the MDI client window.

BeginDeferWindowPos 3.0

Syntax HANDLE BeginDeferWindowPos(nNumWindows)
function BeginDeferWindowPos(NumWindows: Integer): THandle;

This function allocates memory to contain a multiple window-position data structure and returns a handle to the structure. The **DeferWindowPos** function fills this data structure with information about the target position for a window that is about to be moved. The **EndDeferWindowPos**

function accepts this data structure and instantaneously repositions the windows using the information stored in the structure.

Parameters *nNumWindows* **int** Specifies the initial number of windows for which position information is to be stored in the data structure. The **Defer-WindowPos** function increases the size of the structure if needed.

Return value The return value identifies the multiple window-position data structure. The return value is NULL if system resources are not available to allocate the structure.

BeginPaint

Syntax HDC BeginPaint(hWnd, lpPaint)
function BeginPaint(Wnd: HWnd; var Paint: TPaintStruct): HDC;

This function prepares the given window for painting and fills the paint structure pointed to by the *lpPaint* parameter with information about the painting.

The paint structure contains a handle to the device context for the window, a **RECT** data structure that contains the smallest rectangle that completely encloses the update region, and a flag that specifies whether or not the background has been erased.

The **BeginPaint** function automatically sets the clipping region of the device context to exclude any area outside the update region. The update region is set by the **InvalidateRect** or **InvalidateRgn** functions and by the system after sizing, moving, creating, scrolling, or any other operation that affects the client area. If the update region is marked for erasing, **BeginPaint** sends a WM_ERASEBKGND message to the window.

An application should not call the **BeginPaint** function except in response to a WM_PAINT message. Each **BeginPaint** call must have a matching call to the **EndPaint** function.

Parameters

hWnd	**HWND** Identifies the window to be repainted.
lpPaint	**LPPAINTSTRUCT** Points to the **PAINTSTRUCT** data structure that is to receive painting information, such as the device context for the window and the update rectangle.

Return value The return value identifies the device context for the specified window.

Comments If the caret is in the area to be painted, the **BeginPaint** function automatically hides the caret to prevent it from being erased.

BitBlt

Syntax BOOL BitBlt(hDestDC, X, Y, nWidth, nHeight, hSrcDC, XSrc, YSrc, dwRop)
function BitBlt(DestDC: HDC; X, Y, Width, Height: Integer; SrcDC: HDC; XSrc, YSrc: Integer; Rop: Longint): Bool;

This function moves a bitmap from the source device given by the *hSrcDCd* parameter to the destination device given by the *hDestDC* parameter. The *XSrc* and *YSrc* parameters specify the origin on the source device of the bitmap that is to be moved. The *X*, *Y*, *nWidth*, and *nHeight* parameters specify the origin, width, and height of the rectangle on the destination device that is to be filled by the bitmap. The *dwRop* parameter (raster operation) defines how the bits of the source and destination are combined.

Parameters

hDestDC	**HDC** Identifies the device context that is to receive the bitmap.
X	**int** Specifies the logical *x*-coordinate of the upper-left corner of the destination rectangle.
Y	**int** Specifies the logical *y*-coordinate of the upper-left corner of the destination rectangle.
nWidth	**int** Specifies the width (in logical units) of the destination rectangle and source bitmap.
nHeight	**int** Specifies the height (in logical units) of the destination rectangle and source bitmap.
hSrcDC	**HDC** Identifies the device context from which the bitmap will be copied. It must be NULL if the *dwRop* parameter specifies a raster operation that does not include a source.
XSrc	**int** Specifies the logical *x*-coordinate of the upper-left corner of the source bitmap.
YSrc	**int** Specifies the logical *y*-coordinate of the upper-left corner of the source bitmap.
dwRop	**DWORD** Specifies the raster operation to be performed. Raster-operation codes define how the graphics device

interface (GDI) combines colors in output operations that involve a current brush, a possible source bitmap, and a destination bitmap. For a list of raster-operation codes, see Table 4.1, "Raster operations."

Return value The return value specifies whether the bitmap is drawn. It is nonzero if the bitmap is drawn. Otherwise, it is zero.

Comments GDI transforms the *nWidth* and *nHeight* parameters, once by using the destination display context, and once by using the source display context. If the resulting extents do not match, GDI uses the **StretchBlt** function to compress or stretch the source bitmap as necessary. If destination, source, and pattern bitmaps do not have the same color format, the **BitBlt** function converts the source and pattern bitmaps to match the destination. The foreground and background colors of the destination are used in the conversion.

If **BitBlt** converts monochrome bitmaps to color, it sets white bits (1) to the background color and black bits (0) to the foreground color. The foreground and background colors of the destination device context are used. To convert color to monochrome, **BitBlt** sets pixels that match the background color to white (1), and sets all other pixels to black (0). The foreground and background colors of the color-source device context are used.

The foreground color is the current text color for the specified device context, and the background color is the current background color for the specified device context.

Not all devices support the **BitBlt** function. For more information, see the RC_BITBLT raster capability in the **GetDeviceCaps** function, later in this chapter.

Table 4.1 lists the various raster-operation codes for the *dwRop* parameter:

Table 4.1
Raster operations

Code	Description
BLACKNESS	Turns all output black.
DSTINVERT	Inverts the destination bitmap.
MERGECOPY	Combines the pattern and the source bitmap using the Boolean AND operator.
MERGEPAINT	Combines the inverted source bitmap with the destination bitmap using the Boolean OR operator.
NOTSRCCOPY	Copies the inverted source bitmap to the destination.
NOTSRCERASE	Inverts the result of combining the destination and source bitmaps using the Boolean OR operator.
PATCOPY	Copies the pattern to the destination bitmap.

Table 4.1: Raster operations (continued)

PATINVERT	Combines the destination bitmap with the pattern using the Boolean XOR operator.
PATPAINT	Combines the inverted source bitmap with the pattern using the Boolean OR operator. Combines the result of this operation with the destination bitmap using the Boolean OR operator.
SRCAND	Combines pixels of the destination and source bitmaps using the Boolean AND operator.
SRCCOPY	Copies the source bitmap to the destination bitmap.
SRCERASE	Inverts the destination bitmap and combines the result with the source bitmap using the Boolean AND operator.
SRCINVERT	Combines pixels of the destination and source bitmaps using the Boolean XOR operator.
SRCPAINT	Combines pixels of the destination and source bitmaps using the Boolean OR operator.
WHITENESS	Turns all output white.

For a complete list of the raster-operation codes, see Chapter 11, "Binary and ternary raster-operation codes," in Reference, Volume 2.

BringWindowToTop

Syntax void BringWindowToTop(hWnd)
procedure BringWindowToTop(Wnd: HWnd);

This function brings a pop-up or child window to the top of a stack of overlapping windows. In addition, it activates pop-up and top-level windows. The **BringWindowToTop** function should be used to uncover any window that is partially or completely obscured by any overlapping windows.

Parameters *hWnd* **HWND** Identifies the pop-up or child window that is to be brought to the top.

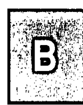
Return value None.

BuildCommDCB

Syntax int BuildCommDCB(lpDef, lpDCB)
function BuildCommDCB(Def: PChar; var DCB: TDCB): Integer;

This function translates the definition string specified by the lpDef parameter into appropriate device-control block codes and places these codes into the block pointed to by the lpDCB parameter.

Parameters *lpDef* **LPSTR** Points to a null-terminated character string that specifies the device-control information for a device. The



	string must have the same form as the DOS **MODE** command-line parameter.
lpDCB	**DCB FAR** *Points to the **DCB** data structure that is to receive the translated string. The structure defines the control setting for the serial-communication device.

Return value The return value specifies the result of the function. It is zero if the string is translated. It is negative if an error occurs.

Comments The **BuildCommDCB** function only fills the buffer. An application should call **SetCommState** to apply these settings to the port. Also, by default, **BuildCommDCB** specifies Xon/Xoff and hardware flow control as disabled. An application should set the appropriate fields in the **DCB** data structure to enable flow control.

CallMsgFilter

Syntax BOOL CallMsgFilter(lpMsg, nCode)
function CallMsgFilter(var Msg: TMsg; Code: Integer): Bool;

This function passes the given message and code to the current message filter function. The message filter function is an application-specified function that examines and modifies all messages. An application specifies the function by using the **SetWindowsHook** function.

Parameters

lpMsg	**LPMSG** Points to an **MSG** data structure that contains the message to be filtered.
nCode	**int** Specifies a code used by the filter function to determine how to process the message.

Return value The return value specifies the state of message processing. It is FALSE if the message should be processed. It is TRUE if the message should not be processed further.

Comments The **CallMsgFilter** function is usually called by Windows to let applications examine and control the flow of messages during internal processing in menus and scroll bars or when moving or sizing a window.

Values given for the *nCode* parameter must not conflict with any of the MSGF_ and HC_ values passed by Windows to the message filter function.

CallWindowProc

Syntax LONG CallWindowProc(lpPrevWndFunc, hWnd, wMsg, wParam, lParam)
function CallWindowProc(PrevWndFunc: TFarProc; Wnd: HWnd; Msg, wParam: Word; lParam: Longint): Longint;

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

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

Parameters

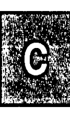
lpPrevWndFunc	**FARPROC** Is the procedure-instance address of the previous window function.
hWnd	**HWND** Identifies the window that receives the message.
wMsg	**WORD** Specifies the message number.
wParam	**WORD** Specifies additional message-dependent information.
lParam	**DWORD** Specifies additional message-dependent information.

Return value The return value specifies the result of the message processing. The possible return values depend on the message sent.

Catch

Syntax int Catch(lpCatchBuf)
function Catch(var CatchBuf: TCatchBuf): Integer;

This function catches the current execution environment and copies it to the buffer pointed to by the *lpCatchBuf* parameter. The execution environment is the state of all system registers and the instruction counter.



Parameters *lpCatchBuf* — **LPCATCHBUF** Points to the **CATCHBUF** structure that will receive the execution environment.

Return value The return value specifies whether the execution environment is copied to the buffer. It is zero if the environment is copied to the buffer.

Comments The **Throw** function uses the buffer to restore the execution environment to its previous values.

The **Catch** function is similar to the C run-time **setjmp** function (which is incompatible with the Windows environment).

ChangeClipboardChain

Syntax

```
BOOL ChangeClipboardChain(hWnd, hWndNext)
function ChangeClipboardChain(Wnd, WndNext: HWnd): Bool;
```

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

Parameters

hWnd — **HWND** Identifies the window that is to be removed from the chain. The handle must previously have been passed to the **SetClipboardViewer** function.

hWndNext — **HWND** Identifies the window that follows *hWnd* in the clipboard-viewer chain (this is the handle returned by the **SetClipboardViewer** function, unless the sequence was changed in response to a WM_CHANGECBCHAIN message).

Return value The return value specifies the status of the *hWnd* window. It is nonzero if the window is found and removed. Otherwise, it is zero.

ChangeMenu

The Microsoft Windows version 3.0 SDK has replaced this function with five specialized functions. These new functions are:

Function	Description
AppendMenu	Appends a menu item to the end of a menu
DeleteMenu	Deletes a menu item from a menu, destroying the menu item
InsertMenu	Inserts a menu item into a menu

ModifyMenu	Modifies a menu item in a menu
RemoveMenu	Removes a menu item from a menu but does not destroy the menu item

Applications written for SDK versions 2.1 and earlier may continue to call **ChangeMenu** as previously documented. New applications should call the new functions listed here.

ChangeSelector 3.0

Syntax WORD ChangeSelector(wDestSelector, wSourceSelector)
function ChangeSelector(DestSelector, SourceSelector:Word):Word;

This function generates a code selector that corresponds to a given data selector, or a data selector that corresponds to a given code selector.

The *wSourceSelector* parameter specifies the selector to be copied and converted; the *wDestSelector* parameter is a selector previously allocated by a call to the **AllocSelector** function. **ChangeSelector** modifies the destination selector to have the same properties as the source selector, but with the opposite code or data attribute. This function changes only the attributes of the selector, not the value of the selector.

Parameters

wDestSelector	**WORD** Specifies a selector previously allocated by **AllocSelector** that receives the converted selector.
wSourceSelector	**WORD** Specifies the selector to be converted.

Return value The return value is the copied and converted selector. It is zero if the function failed.

Comments Windows does not attempt to track changes to the source selector. Consequently, the application should use the converted destination selector immediately after it is returned by this function before any movement of memory can occur.

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

CheckDlgButton

Syntax void CheckDlgButton(hDlg, nIDButton, wCheck)
procedure CheckDlgButton(Dlg: HWnd; IDButton: Integer; Check: Word);

This function places a checkmark next to or removes a checkmark from a button control, or changes the state of a three-state button. The **CheckDlgButton** function sends a BM_SETCHECK message to the button control that has the specified ID in the given dialog box.

Parameters

hDlg	**HWND** Identifies the dialog box that contains the button.
nIDButton	**int** Specifies the button control to be modified.
wCheck	**WORD** Specifies the action to take. If the *wCheck* parameter is nonzero, the **CheckDlgButton** function places a checkmark next to the button; if zero, the checkmark is removed. For three-state buttons, if *wCheck* is 2, the button is grayed; if *wCheck* is 1, it is checked; if *wCheck* is 0, the checkmark is removed.

Return value None.

CheckMenuItem

Syntax BOOL CheckMenuItem(hMenu, wIDCheckItem, wCheck)
function CheckMenuItem(Menu: HMenu; IDCheckItem, Check: Word): Bool;

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

Parameters

hMenu	**HMENU** Identifies the menu.
wIDCheckItem	**WORD** Specifies the menu item to be checked.
wCheck	**WORD** Specifies how to check the menu item and how to determine the item's position in the menu. The *wCheck* parameter can be a combination of the MF_CHECKED or MF_UNCHECKED with MF_BYPOSITION or MF_BYCOMMAND flags. These flags can be combined by using the bitwise OR operator. They have the following meanings:

Value	Meaning
MF_BYCOMMAND	Specifies that the *wIDCheckItem* parameter gives the menu-item ID (MF_BYCOMMAND is the default).
MF_BYPOSITION	Specifies that the *wIDCheckItem* parameter gives the position of the

	menu item (the first item is at position zero).
MF_CHECKED	Adds checkmark.
MF_UNCHECKED	Removes checkmark.

Return value The return value specifies the previous state of the item. It is either MF_CHECKED or MF_UNCHECKED. The return value is –1 if the menu item does not exist.

Comments The *wIDCheckItem* parameter may identify a pop-up menu item as well as a menu item. No special steps are required to check a pop-up menu item.

Top-level menu items cannot be checked.

A pop-up menu item should be checked by position since it does not have a menu-item identifier associated with it.

CheckRadioButton

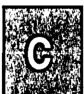
Syntax void CheckRadioButton(hDlg, nIDFirstButton, nIDLastButton, nIDCheckButton)
procedure CheckRadioButton(Dlg: HWnd; IDFirstButton, IDLastButton, IDCheckButton: Integer);

This function checks the radio button specified by the *nIDCheckButton* parameter and removes the checkmark from all other radio buttons in the group of buttons specified by the *nIDFirstButton* and *nIDLastButton* parameters. The **CheckRadioButton** function sends a BM_SETCHECK message to the radio-button control that has the specified ID in the given dialog box.

Parameters

hDlg	**HWND** Identifies the dialog box.
nIDFirstButton	**int** Specifies the integer identifier of the first radio button in the group.
nIDLastButton	**int** Specifies the integer identifier of the last radio button in the group.
nIDCheckButton	**int** Specifies the integer identifier of the radio button to be checked.

Return value None.



ChildWindowFromPoint

Syntax HWNDChildWindowFromPoint(hWndParent, Point)
function ChildWindowFromPoint(Wnd HWnd; APoint: TPoint): HWnd;

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

Parameters

hWndParent	**HWND** Identifies the parent window.
Point	**POINT** Specifies the client coordinates of the point to be tested.

Return value The return value identifies the child window that contains the point. It is NULL if the given point lies outside the parent window. If the point is within the parent window but is not contained within any child window, the handle of the parent window is returned.

Chord

Syntax BOOL Chord(hDC, X1, Y1, X2, Y2, X3, Y3, X4, Y4)
function Chord(DC: HDC; X1, Y1, X2, Y2, X3, Y3, X4, Y4: Integer): Bool;

This function draws a chord (a region bounded by the intersection of an ellipse and a line segment). The *(X1, Y1)* and *(X2, Y2)* parameters specify the upper-left and lower-right corners, respectively, of a rectangle bounding the ellipse that is part of the chord. The *(X3, Y3)* and *(X4, Y4)* parameters specify the endpoints of a line that intersects the ellipse. The chord is drawn by using the selected pen and filled by using the selected brush.

Parameters

hDC	**HDC** Identifies the device context in which the chord will appear.
X1	**int** Specifies the *x*-coordinate of the bounding rectangle's upper-left corner.
Y1	**int** Specifies the *y*-coordinate of the bounding rectangle's upper-left corner.
X2	**int** Specifies the *x*-coordinate of the bounding rectangle's lower-right corner.
Y2	**int** Specifies the *y*-coordinate of the bounding rectangle's lower-right corner.

X3	**int** Specifies the *x*-coordinate of one end of the line segment.
Y3	**int** Specifies the *y*-coordinate of one end of the line segment.
X4	**int** Specifies the *x*-coordinate of one end of the line segment.
Y4	**int** Specifies the *y*-coordinate of one end of the line segment.

Return value The return value specifies whether or not the arc is drawn. It is nonzero if the arc is drawn. Otherwise, it is zero.

ClearCommBreak

Syntax intClearCommBreak(nCid)
function ClearCommBreak(Cid: Integer): Integer;

This function restores character transmission and places the transmission line in a nonbreak state.

Parameters

nCid	**int** Specifies the communication device to be restored. The **OpenComm** function returns this value.

Return value The return value specifies the result of the function. It is zero if the function is successful. It is negative if the *nCid* parameter is not a valid device.

ClientToScreen

Syntax void ClientToScreen(hWnd, lpPoint)
procedure ClientToScreen(Wnd: HWnd; var Point: TPoint);

This function converts the client coordinates of a given point on the display to screen coordinates. The **ClientToScreen** function uses the client coordinates in the **POINT** data structure, pointed to by the *lpPoint* parameter, to compute new screen coordinates; it then replaces the coordinates in the structure with the new coordinates. The new screen coordinates are relative to the upper-left corner of the system display.

Parameters

hWnd	**HWND** Identifies the window whose client area will be used for the conversion.

lpPoint — **LPPOINT** Points to a **POINT** data structure that contains the client coordinates to be converted.

Return value None.

Comments The **ClientToScreen** function assumes that the given point is in client coordinates and is relative to the given window.

ClipCursor

Syntax void ClipCursor(lpRect)
procedure ClipCursor(Rect: PRect);

This function confines the cursor to the rectangle on the display screen given by the *lpRect* parameter. If a subsequent cursor position, given with the **SetCursorPos** function or the mouse, lies outside the rectangle, Windows automatically adjusts the position to keep the cursor inside. If *lpRect* is NULL, the cursor is free to move anywhere on the display screen.

Parameters *lpRect* — **LPRECT** Points to a **RECT** data structure that contains the screen coordinates of the upper-left and lower-right corners of the confining rectangle.

Return value None.

Comments The cursor is a shared resource. An application that has confined the cursor to a given rectangle must free it before relinquishing control to another application.

CloseClipboard

Syntax BOOL CloseClipboard( )
function CloseClipboard: Bool;

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

Parameters None.

Return value The return value specifies whether the clipboard is closed. It is nonzero if the clipboard is closed. Otherwise, it is zero.

CloseComm

Syntax int CloseComm(nCid)
function CloseComm(Cid: Integer): Integer;

This function closes the communication device specified by the *nCid* parameter and frees any memory allocated for the device's transmit and receive queues. All characters in the output queue are sent before the communication device is closed.

Parameters *nCid* **int** Specifies the device to be closed. The **OpenComm** function returns this value.

Return value The return value specifies the result of the function. It is zero if the device is closed. It is negative if an error occurred.

CloseMetaFile

Syntax HANDLE CloseMetaFile(hDC)
function CloseMetaFile(DC: THandle): THandle;

This function closes the metafile device context and creates a metafile handle that can be used to play the metafile by using the **PlayMetaFile** function.

Parameters *hDC* **HANDLE** Identifies the metafile device context to be closed.

Return value The return value identifies the metafile if the function is successful. Otherwise, it is NULL.

CloseSound

Syntax void CloseSound( )
procedure CloseSound;

This function closes access to the play device and frees the device for opening by other applications. The **CloseSound** function flushes all voice queues and frees any buffers allocated for these queues.

Parameters None.

Return value None.

C

CloseWindow

Syntax void CloseWindow(hWnd)
procedure CloseWindow(Wnd: HWnd);

This function minimizes the specified window. If the window is an overlapped window, it is minimized by removing the client area and caption of the open window from the display screen and moving the window's icon into the icon area of the screen.

Parameters *hWnd* **HWND** Identifies the window to be minimized.

Return value None.

Comments This function has no effect if the *hWnd* parameter is a handle to a pop-up or child window.

CombineRgn

Syntax int CombineRgn(hDestRgn, hSrcRgn1, hSrcRgn2, nCombineMode)
function CombineRgn(DestRgn, SrcRgn1, SrcRgn2: HRgn; CombineMode: Integer): Integer;

This function creates a new region by combining two existing regions. The method used to combine the regions is specified by the *nCombineMode* parameter.

Parameters

hDestRgn **HRGN** Identifies an existing region that will be replaced by the new region.

hSrcRgn1 **HRGN** Identifies an existing region.

hSrcRgn2 **HRGN** Identifies an existing region.

nCombineMode **int** Specifies the operation to be performed on the two existing regions. It can be any one of the following values:

Value	Meaning
RGN_AND	Uses overlapping areas of both regions (intersection).
RGN_COPY	Creates a copy of region 1 (identified by *hSrcRgn1*).
RGN_DIFF	Saves the areas of region 1 (identified by the *hSrcRgn1* parameter) that are not part of

	region 2 (identified by the *hSrcRgn2* parameter).
RGN_OR	Combines all of both regions (union).
RGN_XOR	Combines both regions but removes overlapping areas.

Return value The return value specifies the type of the resulting region. It can be any one of the following values:

Value	Meaning
COMPLEXREGION	New region has overlapping borders.
ERROR	No new region created.
NULLREGION	New region is empty.
SIMPLEREGION	New region has no overlapping borders.

Comments If the *hDestRgn* parameter does not identify an existing region, the application must pass a far pointer to a previously allocated **HRGN** as the *hDestRgn* parameter.

CopyMetaFile

Syntax HANDLE CopyMetaFile(hSrcMetaFile, lpFilename)
function CopyMetaFile(SrcMetaFile: THandle; FileName: PChar): THandle;

This function copies the source metafile to the file pointed to by the *lpFilename* parameter and returns a handle to the new metafile. If *lpFilename* is NULL, the source is copied to a memory metafile.

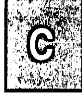
Parameters

hSrcMetaFile	**HANDLE** Identifies the source metafile.
lpFilename	**LPSTR** Points to a null-terminated character string that specifies the file that is to receive the metafile.

Return value The return value identifies the new metafile.

CopyRect

Syntax int CopyRect(lpDestRect, lpSourceRect)
procedure CopyRect(var DestRect, SourceRect: TRect);



This function copies the rectangle pointed to by the *lpSourceRect* parameter to the **RECT** data structure pointed to by the *lpDestRect* parameter.

Parameters

lpDestRect	**LPRECT** Points to a **RECT** data structure.
lpSourceRect	**LPRECT** Points to a **RECT** data structure.

Return value Although the **CopyRect** function return type is an integer, the return value is not used and has no meaning.

CountClipboardFormats

Syntax int CountClipboardFormats( )
function CountClipboardFormats: Integer;

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

Parameters None.

Return value The return value specifies the number of data formats in the clipboard.

CountVoiceNotes

Syntax int CountVoiceNotes(nVoice)
function CountVoiceNotes(Voice: Integer): Integer;

This function retrieves a count of the number of notes in the specified queue. Only those queue entries that result from calls to the **SetVoiceNote** function are counted.

Parameters

nVoice	**int** Specifies the voice queue to be counted. The first voice queue is numbered 1.

Return value The return value specifies the number of notes in the given queue.

CreateBitmap

Syntax HBITMAP CreateBitmap(nWidth, nHeight, nPlanes, nBitCount, lpBits)
function CreateBitmap(Width, Height: Integer; Planes, BitCount: Byte; Bits: Pointer): HBitmap;

This function creates a device-dependent memory bitmap that has the specified width, height, and bit pattern. The bitmap can subsequently be selected as the current bitmap for a memory display by using the **SelectObject** function.

Although a bitmap cannot be copied directly to a display device, the **BitBlt** function can copy it from a memory display context (in which it is the current bitmap) to any compatible device.

Parameters

nWidth	**int** Specifies the width (in pixels) of the bitmap.
nHeight	**int** Specifies the height (in pixels) of the bitmap.
nPlanes	**BYTE** Specifies the number of color planes in the bitmap. Each plane has *nWidth* × *nHeight* × *nBitCount* bits.
nBitCount	**BYTE** Specifies the number of color bits per display pixel.
lpBits	**LPSTR** Points to a short-integer array that contains the initial bitmap bit values. If it is NULL, the new bitmap is left uninitialized. For more information, see the description of the **bmBits** field in the **BITMAP** data structure in Chapter 7, "Data types and structures," in *Reference, Volume 2*.

Return value The return value identifies a bitmap if the function is successful. Otherwise, it is NULL.

CreateBitmapIndirect

Syntax HBITMAP CreateBitmapIndirect(lpBitmap)
function CreateBitmapIndirect(var Bitmap: TBitmap): HBitmap;

This function creates a bitmap that has the width, height, and bit pattern given in the data structure pointed to by the *lpBitmap* parameter. Although a bitmap cannot be directly selected for a display device, it can be selected as the current bitmap for a memory display and copied to any compatible display device by using the **BitBlt** function.

Parameters

lpBitmap	**BITMAP FAR *** Points to a **BITMAP** data structure that contains information about the bitmap.

Return value The return value identifies a bitmap if the function is successful. Otherwise, it is NULL.

C

CreateBrushIndirect

Syntax HBRUSH CreateBrushIndirect(lpLogBrush)
function CreateBrushIndirect(var LogBrush: TLogBrush): HBrush;

This function creates a logical brush that has the style, color, and pattern given in the data structure pointed to by the *lpLogBrush* parameter. The brush can subsequently be selected as the current brush for any device.

Parameters *lpLogBrush* **LOGBRUSH FAR** * Points to a **LOGBRUSH** data structure that contains information about the brush.

Return value The return value identifies a logical brush if the function is successful. Otherwise, it is NULL.

Comments A brush created using a monochrome (one plane, one bit per pixel) bitmap is drawn using the current text and background colors. Pixels represented by a bit set to 0 will be drawn with the current text color, and pixels represented by a bit set to 1 will be drawn with the current background color.

CreateCaret

Syntax void CreateCaret(hWnd, hBitmap, nWidth, nHeight)
procedure CreateCaret(Wnd: HWnd; Bitmap: HBitmap; Width, Height: Integer);

This function creates a new shape for the system caret and assigns ownership of the caret to the given window. The caret shape can be a line, block, or bitmap as defined by the *hBitmap* parameter. If *hBitmap* is a bitmap handle, the *nWidth* and *nHeight* parameters are ignored; the bitmap defines its own width and height. (The bitmap handle must have been previously created by using the **CreateBitmap**, **CreateDIBitmap**, or **LoadBitmap** function.) If *hBitmap* is NULL or 1, *nWidth* and *nHeight* give the caret's width and height (in logical units); the exact width and height (in pixels) depend on the window's mapping mode.

If *nWidth* or *nHeight* is zero, the caret width or height is set to the system's window-border width or height. Using the window-border width or height guarantees that the caret will be visible on a high-resolution display.

The **CreateCaret** function automatically destroys the previous caret shape, if any, regardless of which window owns the caret. Once created, the caret is initially hidden. To show the caret, the **ShowCaret** function must be called.

Parameters

hWnd	**HWND** Identifies the window that owns the new caret.
hBitmap	**HBITMAP** Identifies the bitmap that defines the caret shape. If *hBitmap* is NULL, the caret is solid; if *hBitmap* is 1, the caret is gray.
nWidth	**int** Specifies the width of the caret (in logical units).
nHeight	**int** Specifies the height of the caret (in logical units).

Return value None.

Comments The system caret is a shared resource. A window should create a caret only when it has the input focus or is active. It should destroy the caret before losing the input focus or becoming inactive.

The system's window-border width or height can be retrieved by using the **GetSystemMetrics** function with the SM_CXBORDER and SM_CYBORDER indexes.

CreateCompatibleBitmap

Syntax HBITMAP CreateCompatibleBitmap(hDC, nWidth, nHeight)
function CreateCompatibleBitmap(DC: HDC; Width, Height: Integer): HBitmap;

This function creates a bitmap that is compatible with the device specified by the *hDC* parameter. The bitmap has the same number of color planes or the same bits-per-pixel format as the specified device. It can be selected as the current bitmap for any memory device that is compatible with the one specified by *hDC*.

If *hDC* is a memory device context, the bitmap returned has the same format as the currently selected bitmap in that device context. A memory device context is a block of memory that represents a display surface. It can be used to prepare images in memory before copying them to the actual display surface of the compatible device.

When a memory device context is created, GDI automatically selects a monochrome stock bitmap for it.

C

Since a color memory device context can have either color or monochrome bitmaps selected, the format of the bitmap returned by the **CreateCompatibleBitmap** function is not always the same; however, the format of a compatible bitmap for a nonmemory device context is always in the format of the device.

Parameters

hDC	**HDC** Identifies the device context.
nWidth	**int** Specifies the width (in bits) of the bitmap.
nHeight	**int** Specifies the height (in bits) of the bitmap.

Return value The return value identifies a bitmap if the function is successful. Otherwise, it is NULL.

CreateCompatibleDC

Syntax HDCCreateCompatibleDC(hDC)
function CreateCompatibleDC(DC: HDC): HDC;

This function creates a memory device context that is compatible with the device specified by the *hDC* parameter. A memory device context is a block of memory that represents a display surface. It can be used to prepare images in memory before copying them to the actual device surface of the compatible device.

When a memory device context is created, GDI automatically selects a 1-by-1 monochrome stock bitmap for it.

Parameters

hDC	**HDC** Identifies the device context. If *hDC* is NULL, the function creates a memory device context that is compatible with the system display.

Return value The return value identifies the new memory device context if the function is successful. Otherwise, it is NULL.

Comments This function can only be used to create compatible device contexts for devices that support raster operations. For more information, see the RC_BITBLT raster capability in the **GetDeviceCaps** function, later in this chapter.

GDI output functions can be used with a memory device context only if a bitmap has been created and selected into that context.

When the application no longer requires the device context, it should free it by calling the **DeleteDC** function.

CreateCursor 3.0

Syntax HCURSOR CreateCursor(hInstance, nXhotspot, nYhotspot, nWidth, nHeight, lpANDbitPlane, lpXORbitPlane)
function CreateCursor(Instance: THandle; Xhotspot, Yhotspot, Width, Height: Integer; ANDBitPlane, XORBitPlane: Pointer): HCursor;

This function creates a cursor that has specified width, height, and bit patterns.

Parameters

hInstance	**HANDLE** Identifies an instance of the module creating the cursor.
nXhotspot	**int** Specifies the horizontal position of the cursor hotspot.
nYhotspot	**int** Specifies the vertical position of the cursor hotspot.
nWidth	**int** Specifies the width in pixels of the cursor.
nHeight	**int** Specifies the height in pixels of the cursor.
lpANDbitPlane	**LPSTR** Points to an array of bytes containing the bit values for the AND mask of the cursor. This can be the bits of a device-dependent monochrome bitmap.
lpXORbitPlane	**LPSTR** Points to an array of bytes containing the bit values for the XOR mask of the cursor. This can be the bits of a device-dependent monochrome bitmap.

Return value The return value identifies the cursor if the function was successful. Otherwise, it is NULL.

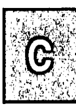
CreateDC

Syntax HDC CreateDC(lpDriverName, lpDeviceName, lpOutput, lpInitData)
function CreateDC(DriverName, DeviceName, Output: PChar; InitData: Pointer): HDC;

This function creates a device context for the specified device. The *lpDriverName*, *lpDeviceName*, and *lpOutput* parameters specify the device driver, device name, and physical output medium (file or port), respectively.

Parameters

lpDriverName	**LPSTR** Points to a null-terminated character string that specifies the DOS filename (without extension) of the device driver (for example, Epson ©).



lpDeviceName	**LPSTR** Points to a null-terminated character string that specifies the name of the specific device to be supported (for example, Epson FX-80). The *lpDeviceName* parameter is used if the module supports more than one device.
lpOutput	**LPSTR** Points to a null-terminated character string that specifies the DOS file or device name for the physical output medium (file or output port).
lpInitData	**LPDEVMODE** Points to a **DEVMODE** data structure containing device-specific initialization data for the device driver. The **ExtDeviceMode** retrieves this structure filled in for a given device. The *lpInitData* parameter must be NULL if the device driver is to use the default initialization (if any) specified by the user through the Control Panel.

Return value The return value identifies a device context for the specified device if the function is successful. Otherwise, it is NULL.

Comments DOS device names follow DOS conventions; an ending colon (:) is recommended, but optional. Windows strips the terminating colon so that a device name ending with a colon is mapped to the same port as the same name without a colon. The driver and port names must not contain leading or trailing spaces.

CreateDialog

Syntax HWND CreateDialog(hInstance, lpTemplateName, hWndParent, lpDialogFunc)
function (Instance: THandle; TemplateName: PChar; WndParent: HWnd; DialogFunc: TFarProc): HWnd;

This function creates a modeless dialog box that has the size, style, and controls defined by the dialog-box template given by the *lpTemplateName* parameter. The *hWndParent* parameter identifies the application window that owns the dialog box. The dialog function pointed to by the *lpDialogFunc* parameter processes any messages received by the dialog box.

The **CreateDialog** function sends a WM_INITDIALOG message to the dialog function before displaying the dialog box. This message allows the dialog function to initialize the dialog-box controls.

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

Parameters

hInstance	**HANDLE** Identifies an instance of the module whose executable file contains the dialog-box template.
lpTemplateName	**LPSTR** Points to a character string that names the dialog-box template. The string must be a null-terminated character string.
hWndParent	**HWND** Identifies the window that owns the dialog box.
lpDialogFunc	**FARPROC** Is the procedure-instance address for the dialog function. See the following "Comments" section for details.

Return value The return value is the window handle of the dialog box. It is NULL if the function cannot create the dialog box.

Comments Use the WS_VISIBLE style for the dialog-box template if the dialog box should appear in the parent window upon creation.

Use the **DestroyWindow** function to destroy a dialog box created by the **CreateDialog** function.

A dialog box can contain up to 255 controls.
The callback function must use the Pascal calling convention and must be declared **FAR**.

Callback function

BOOL FAR PASCAL *DialogFunc(hDlg, wMsg, wParam, lParam)*

HWND *hDlg;*

WORD *wMsg;*

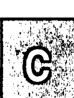
WORD *wParam;*

DWORD *lParam;*

DialogFunc is a placeholder for the application-supplied function name. The actual name must be exported by including it in an **EXPORTS** statement in the application's module-definition file.

Parameters

hDlg	Identifies the dialog box that receives the message.
wMsg	Specifies the message number.



wParam	Specifies 16 bits of additional message-dependent information.
lParam	Specifies 32 bits of additional message-dependent information.

Return value Except in response to the WM_INITDIALOG message, the dialog function should return nonzero if the function processes the message, and zero if it does not. In response to a WM_INITDIALOG message, the dialog function should return zero if it calls the **SetFocus** function to set the focus to one of the controls in the dialog box. Otherwise, it should return nonzero, in which case Windows will set the focus to the first control in the dialog box that can be given the focus.

Comments The dialog function is used only if the dialog class is used for the dialog box. This is the default class and is used if no explicit class is given in the dialog-box template. Although the dialog function is similar to a window function, it must not call the **DefWindowProc** function to process unwanted messages. Unwanted messages are processed internally by the dialog-class window function.

The dialog-function address, passed as the *lpDialogFunc* parameter, must be created by using the **MakeProcInstance** function.

CreateDialogIndirect

Syntax HWND CreateDialogIndirect(hInstance, lpDialogTemplate, hWndParent, lpDialogFunc)
function CreateDialogIndirect(Instance: THandle; DialogTemplate: Pointer; WndParent: HWnd; DialogFunc: TFarProc): HWnd;

This function creates a modeless dialog box that has the size, style, and controls defined by the dialog-box template given by the *lpDialogTemplate* parameter. The *hWndParent* parameter identifies the application window that owns the dialog box. The dialog function pointed to by the *lpDialogFunc* parameter processes any messages received by the dialog box.

The **CreateDialogIndirect** function sends a WM_INITDIALOG message to the dialog function before displaying the dialog box. This message allows the dialog function to initialize the dialog-box controls.

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

Parameters

hInstance	**HANDLE** Identifies an instance of the module whose executable file contains the dialog-box template.
lpDialogTemplate	**LPSTR** Points to a block of memory that contains a **DLGTEMPLATE** data structure.
hWndParent	**HWND** Identifies the window that owns the dialog box.
lpDialogFunc	**FARPROC** Is the procedure-instance address of the dialog function. See the following "Comments" section for details.

Return value The return value is the window handle of the dialog box. It is NULL if the function cannot create either the dialog box or any controls in the dialog box.

Comments Use the WS_VISIBLE style in the dialog-box template if the dialog box should appear in the parent window upon creation.

A dialog box can contain up to 255 controls.
The callback function must use the Pascal calling convention and must be declared **FAR**.

Callback function

BOOL FAR PASCAL *DialogFunc(hDlg, wMsg, wParam, lParam)*
HWND *hDlg;*
WORD *wMsg;*
WORD *wParam;*
DWORD *lParam;*

DialogFunc is a placeholder for the application-supplied function name. The actual name must be exported by including it in an **EXPORTS** statement in the application's module-definition file.

Parameters

hDlg	Identifies the dialog box that receives the message.
wMsg	Specifies the message number.
wParam	Specifies 16 bits of additional message-dependent information.
lParam	Specifies 32 bits of additional message-dependent information.

Return value Except in response to the WM_INITDIALOG message, the dialog function should return nonzero if the function processes the message, and zero if it

C

does not. In response to a WM_INITDIALOG message, the dialog function should return zero if it calls the **SetFocus** function to set the focus to one of the controls in the dialog box. Otherwise, it should return nonzero, in which case Windows will set the focus to the first control in the dialog box that can be given the focus.

Comments The dialog function is used only if the dialog class is used for the dialog box. This is the default class and is used if no explicit class is given in the dialog-box template. Although the dialog function is similar to a window function, it must not call the **DefWindowProc** function to process unwanted messages. Unwanted messages are processed internally by the dialog-class window function.

The dialog-function address, passed as the *lpDialogFunc* parameter, must be created by using the **MakeProcInstance** function.

CreateDialogIndirectParam 3.0

Syntax HWND CreateDialogIndirectParam(hInstance, lpDialogTemplate, hWndParent, lpDialogFunc, dwInitParam)
function CreateDialogIndirectParam (Instance: Thandle; DialogTemplate; WndParent: HWnd; DialogFunc: TFarProc; InitParam: Longint): HWnd;

This function creates a modeless dialog box, sends a WM_INITDIALOG message to the dialog function before displaying the dialog box, and passes *dwInitParam* as the message *lParam*. This message allows the dialog function to initialize the dialog-box controls. Otherwise, this function is identical to the **CreateDialogIndirect** function.

For more information on creating a modeless dialog box, see the description of the **CreateDialogIndirect** function.

Parameters

hInstance **HANDLE** Identifies an instance of the module whose executable file contains the dialog-box template.

lpDialogTemplate
LPSTR Points to a block of memory that contains a **DLGTEMPLATE** data structure.

hWndParent **HWND** Identifies the window that owns the dialog box.

lpDialogFunc **FARPROC** Is the procedure-instance address of the dialog function. For details, see the "Comments" section in the description of the **CreateDialogIndirect** function.

dwInitParam **DWORD** Is a 32-bit value which **CreateDialogIndirectParam** passes to the dialog function when it creates the dialog box.

Return value The return value is the window handle of the dialog box. It is NULL if the function cannot create either the dialog box or any controls in the dialog box.

CreateDialogParam 3.0

Syntax HWND CreateDialogParam(hInstance, lpTemplateName, hWndParent, lpDialogFunc, dwInitParam)
function CreateDialogParam(Instance: THandle; TemplateName: PChar; WndParent: HWnd; DialogFunc: TFarProc; InitParam: Longint): HWnd;

This function creates a modeless dialog box, sends a WM_INITDIALOG message to the dialog function before displaying the dialog box, and passes *dwInitParam* as the message *lParam*. This message allows the dialog function to initialize the dialog-box controls. Otherwise, this function is identical to the **CreateDialog** function.

For more information on creating a modeless dialog box, see the description of the **CreateDialog** function.

Parameters

hInstance **HANDLE** Identifies an instance of the module whose executable file contains the dialog-box template.

lpTemplateName **LPSTR** Points to a character string that names the dialog-box template. The string must be a null-terminated character string.

hWndParent **HWND** Identifies the window that owns the dialog box.

lpDialogFunc **FARPROC** Is the procedure-instance address for the dialog function. For details, see the "Comments" section of the **CreateDialog** function.

dwInitParam **DWORD** Is a 32-bit value which **CreateDialogParam** passes to the dialog function when it creates the dialog box.

Return value The return value is the window handle of the dialog box. It is –1 if the function cannot create the dialog box.

C

CreateDIBitmap 3.0

Syntax HBITMAP CreateDIBitmap(hDC, lpInfoHeader, dwUsage, lpInitBits, lpInitInfo, wUsage)
function CreateDIBitmap(DC: HDC; var InfoHeader: TBitmapInfoHeader; dwUsage: Longint; InitBits: PChar; var InitInfo: TBitmapInfo; wUsage: Word): HBitmap;

This function creates a device-specific memory bitmap from a device-independent bitmap (DIB) specification and optionally sets bits in the bitmap.

Parameters

hDC — **HDC** Identifies the device context.

lpInfoHeader — **LPBITMAPINFOHEADER** Points to a **BITMAPINFOHEADER** structure that describes the size and format of the device-independent bitmap.

dwUsage — **DWORD** Indicates whether the memory bitmap is to be initialized. If *dwUsage* is set to CBM_INIT, **CreateDIBitmap** will initialize the bitmap with the bits specified by *lpInitBits* and *lpInitInfo*

lpInitBits — **LPSTR** Points to a byte array that contains the initial bitmap values. The format of the bitmap values depends on the **biBitCount** field of the **BITMAPINFO** structure identified by *lpInitInfo*. See the description of the **BITMAPINFO** data structure in Chapter 7, "Data Types and Structures," in *Reference, Volume 2*, for more information.

lpInitInfo — **LPBITMAPINFO** Points to a **BITMAPINFO** data structure that describes the dimensions and color format of *lpInitBits*.

wUsage — **WORD** Specifies whether the **bmiColors[ ]** fields of the *lpInitInfo* data structure contain explicit RGB values or indexes into the currently realized logical palette. The *wUsage* parameter must be one of the following values:

Value	Meaning
DIB_PAL_COLORS	The color table consists of an array of 16-bit indexes into the currently realized logical palette.

DIB_RGB_COLORS	The color table contains literal RGB values.

Return value The return value identifies a bitmap if the function is successful. Otherwise, it is NULL.

Comments This function also accepts a device-independent bitmap specification formatted for Microsoft OS/2 Presentation Manager versions 1.1 and 1.2 if the *lpInfoHeader* points to a **BITMAPCOREHEADER** data structure and the *lpInitInfo* parameter points to a **BITMAPCOREINFO** data structure.

CreateDIBPatternBrush 3.0

Syntax HBRUSH CreateDIBPatternBrush(hPackedDIB, wUsage)
function CreateDIBPatternBrush(PackedDIB: THandle; Usage: Word): HBrush;

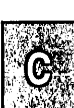
This function creates a logical brush that has the pattern specified by the device-independent bitmap (DIB) defined by the the *hPackedDIB* parameter. The brush can subsequently be selected for any device that supports raster operations. For more information, see the RC_BITBLT raster capability in the **GetDeviceCaps** function, later in this chapter.

Parameters

hPackedDIB **GLOBALHANDLE** Identifies a global memory object containing a packed device-independent bitmap. To obtain this handle, an application calls the **GlobalAlloc** function to allocate a block of global memory and then fills the memory with the packed DIB. A packed DIB consists of a **BITMAPINFO** data structure immediately followed by the array of bytes which define the pixels of the bitmap

wUsage **WORD** Specifies whether the **bmiColors[ ]** fields of the **BITMAPINFO** data structure contain explicit RGB values or indexes into the currently realized logical palette. The *wUsage* parameter must be one of the following values:

Value	Meaning
DIB_PAL_COLORS	The color table contains literal RGB values. into the currently realized logical palette.
DIB_RGB_COLORS	The color table consists of an array of 16-bit indexes.



Return value The return value identifies a logical brush if the function is successful. Otherwise, it is NULL.

Comments When an application selects a two-color DIB pattern brush into a monochrome device context, Windows ignores the colors specified in the DIB and instead displays the pattern brush using the current background and foreground colors of the device context. Pixels mapped to the first color (at offset 0 in the DIB color table) of the DIB are displayed using the foreground color, and pixels mapped to the second color (at offset 1 in the color table) are displayed using the background color. The **SetTextColor** and **SetBkColor** functions change the foreground and background colors, respectively, for a device context.

CreateDiscardableBitmap

Syntax HBITMAP CreateDiscardableBitmap(hDC, nWidth, nHeight)
function CreateDiscardableBitmap(DC: HDC; Width, Height: Integer): HBitmap;

This function creates a discardable bitmap that is compatible with the device identified by the *hDC* parameter. The bitmap has the same number of color planes or the same bits-per-pixel format as the specified device. An application can select this bitmap as the current bitmap for a memory device that is compatible with the one specified by the *hDC* parameter.

Parameters

hDC	**HDC** Identifies a device context.
nWidth	**int** Specifies the width (in bits) of the bitmap.
nHeight	**int** Specifies the height (in bits) of the bitmap.

Return value The return value identifies a bitmap if the function is successful. Otherwise, it is NULL.

Comments Windows can discard a bitmap created by this function only if an application has not selected it into a display context. If Windows discards the bitmap when it is not selected and the application later attempts to select it, the **SelectObject** function will return zero. When this occurs, the application should remove the handle to the bitmap by using **DeleteObject**.

CreateEllipticRgn

Syntax HRGN CreateEllipticRgn(X1, Y1, X2, Y2)
function CreateEllipticRgn(X1, Y1, X2, Y2: Integer): HRgn;

This function creates an elliptical region.

Parameters

X1	**int** Specifies the *x*-coordinate of the upper-left corner of the bounding rectangle of the ellipse.
Y1	**int** Specifies the *y*-coordinate of the upper-left corner of the bounding rectangle of the ellipse.
X2	**int** Specifies the *x*-coordinate of the lower-right corner of the bounding rectangle of the ellipse.
Y2	**int** Specifies the *y*-coordinate of the lower-right corner of the bounding rectangle of the ellipse.

Return value The return value identifies a new region if the function is successful. Otherwise, it is NULL.

Comments The width of the rectangle, specified by the absolute value of *X2* – *X1*, must not exceed 32,767 units. This limit also applies to the height of the rectangle.

CreateEllipticRgnIndirect

Syntax HRGN CreateEllipticRgnIndirect(lpRect)
function CreateEllipticRgnIndirect(var Rect: TRect): HRgn;

This function creates an elliptical region.

Parameters

lpRect	**LPRECT** Points to a **RECT** data structure that contains the coordinates of the upper-left and lower-right corners of the bounding rectangle of the ellipse.

Return value The return value identifies a new region if the function is successful. Otherwise, it is NULL.

Comments The width of the rectangle must not exceed 32,767 units. This limit applies to the height of the rectangle as well.

C

CreateFont

Syntax HFONT CreateFont(nHeight, nWidth, nEscapement, nOrientation, nWeight, cItalic, cUnderline, cStrikeOut, cCharSet, cOutputPrecision, cClipPrecision, cQuality, cPitchAndFamily, lpFacename)

function CreateFont(Height, Width, Escapement, Orientation, Weight: Integer; Italic, Underline, StrikeOut, CharSet, OutputPrecision, ClipPrecision, Quality, PitchAndFamily: Byte; FaceName: PChar): HFont;

This function creates a logical font that has the specified characteristics. The logical font can subsequently be selected as the font for any device.

Parameters

nHeight	**int** Specifies the desired height (in logical units) of the font. The font height can be specified in three ways: If *nHeight* is greater than zero, it is transformed into device units and matched against the cell height of the available fonts. If it is zero, a reasonable default size is used. If it is less than zero, it is transformed into device units and the absolute value is matched against the character height of the available fonts. For all height comparisons, the font mapper looks for the largest font that does not exceed the requested size, and, if there is no such font, looks for the smallest font available.
nWidth	**int** Specifies the average width (in logical units) of characters in the font. If *nWidth* is zero, the aspect ratio of the device will be matched against the digitization aspect ratio of the available fonts to find the closest match, determined by the absolute value of the difference.
nEscapement	**int** Specifies the angle (in tenths of degrees) of each line of text written in the font (relative to the bottom of the page).
nOrientation	**int** Specifies the angle (in tenths of degrees) of each character's baseline (relative to the bottom of the page).
nWeight	**int** Specifies the desired weight of the font in the range 0 to 1000 (for example, 400 is normal, 700 is bold). If *nWeight* is zero, a default weight is used.
cItalic	**BYTE** Specifies whether the font is italic.

cUnderline	**BYTE** Specifies whether the font is underlined.
cStrikeOut	**BYTE** Specifies whether characters in the font are struck out.
cCharSet	**BYTE** Specifies the desired character set. The following values are predefined: ■ ANSI_CHARSET ■ OEM_CHARSET ■ SYMBOL_CHARSET ■ The OEM character set is system-dependent. Fonts with other character sets may exist in the system. If an application uses a font with an unknown character set, it should not attempt to translate or interpret strings that are to be rendered with that font. Instead, the strings should be passed directly to the output device driver.
cOutputPrecision	**BYTE** Specifies the desired output precision. The output precision defines how closely the output must match the requested font's height, width, character orientation, escapement, and pitch. It can be any one of the following values: ■ OUT_CHARACTER_PRECIS ■ OUT_DEFAULT_PRECIS ■ OUT_STRING_PRECIS ■ OUT_STROKE_PRECIS
cClipPrecision	**BYTE** Specifies the desired clipping precision. The clipping precision defines how to clip characters that are partially outside the clipping region. It can be any one of the following values: ■ CLIP_CHARACTER_PRECIS ■ CLIP_DEFAULT_PRECIS ■ CLIP_STROKE_PRECIS
cQuality	**BYTE** Specifies the desired output quality. The output quality defines how carefully GDI must attempt to match the logical-font attributes to those of an actual physical font. It can be any one of the following values: ■ DEFAULT_QUALITY ■ DRAFT_QUALITY ■ PROOF_QUALITY

cPitchAndFamily **BYTE** Specifies the pitch and family of the font. The two low-order bits specify the pitch of the font and can be any one of the following values:

- DEFAULT_PITCH
- FIXED_PITCH
- VARIABLE_PITCH

The four high-order bits of the field specify the font family and can be any one of the following values:

- FF_DECORATIVE
- FF_DONTCARE
- FF_MODERN
- FF_ROMAN
- FF_SCRIPT
- FF_SWISS

lpFacename **LPSTR** Points to a null-terminated character string that specifies the typeface name of the font. The length of this string must not exceed 30 characters. The **EnumFonts** function can be used to enumerate the typeface names of all currently available fonts.

Return value The return value identifies a logical font if the function is successful. Otherwise, it is NULL.

Comments The **CreateFont** function does not create a new font. It merely selects the closest match from the fonts available in GDI's pool of physical fonts.

CreateFontIndirect

Syntax HFONTCreateFontIndirect(lpLogFont)
function CreateFontIndirect(var LogFont: TLogFont): HFont;

This function creates a logical font that has the characteristics given in the data structure pointed to by the *lpLogFont* parameter. The font can subsequently be selected as the current font for any device.

Parameters *lpLogFont* **LOGFONT FAR *** Points to a **LOGFONT** data structure that defines the characteristics of the logical font.

Return value The return value identifies a logical font if the function is successful. Otherwise, it is NULL.

Comments The **CreateFontIndirect** function creates a logical font that has all the specified characteristics. When the font is selected by using the **SelectObject** function, GDI's font mapper attempts to match the logical

font with an existing physical font. If it fails to find an exact font, it provides an alternate whose characteristics match as many of the requested characteristics as possible. For a description of the font mapper, see Chapter 2, "Graphics device interface functions."

CreateHatchBrush

Syntax HBRUSHCreateHatchBrush(nIndex, crColor)
function CreateHatchBrush(Index: Integer; Color: TColorRef): HBrush;

This function creates a logical brush that has the specified hatched pattern and color. The brush can subsequently be selected as the current brush for any device.

Parameters

nIndex **int** Specifies the hatch style of the brush. It can be any one of the following values:

Value	Meaning
HS_BDIAGONAL	45-degree upward hatch (left to right)
HS_CROSS	Horizontal and vertical crosshatch
HS_DIAGCROSS	45-degree crosshatch
HS_FDIAGONAL	45-degree downward hatch (left to right)
HS_HORIZONTAL	Horizontal hatch
HS_VERTICAL	Vertical hatch

crColor **COLORREF** Specifies the foreground color of the brush (the color of the hatches).

Return value The return value identifies a logical brush if the function is successful. Otherwise, it is NULL.

CreateIC

Syntax HDC CreateIC(lpDriverName, lpDeviceName, lpOutput, lpInitData)
function CreateIC(DriverName, DeviceName, Output: PChar; InitDate: Pointer): HDC;

This function creates an information context for the specified device. The information context provides a fast way to get information about the device without creating a device context.

Parameters

lpDriverName	**LPSTR** Points to a null-terminated character string that specifies the DOS filename (without extension) of the device driver (for example, EPSON).
lpDeviceName	**LPSTR** Points to a null-terminated character string that specifies the name of the specific device to be supported (for example, EPSON FX-80). The *lpDeviceName* parameter is used if the module supports more than one device.
lpOutput	**LPSTR** Points to a null-terminated character string that specifies the DOS file or device name for the physical output medium (file or port).
lpInitData	**LPSTR** Points to device-specific initialization data for the device driver. The *lpInitData* parameter must be NULL if the device driver is to use the default initialization (if any) specified by the user through the Control Panel.

Return value The return value identifies an information context for the specified device if the function is successful. Otherwise, it is NULL.

Comments DOS device names follow DOS conventions; an ending colon (:) is recommended, but optional. Windows strips the terminating colon so that a device name ending with a colon is mapped to the same port as the same name without a colon.

The driver and port names must not contain leading or trailing spaces.

GDI output functions cannot be used with information contexts.

CreateIcon 3.0

Syntax HICON CreateIcon(hInstance, nWidth, nHeight, nPlanes, nBitsPixel, lpANDbits, lpXORbits)
function CreateIcon(Instance: THandle; Width, Height: Integer; Planes, BitsPixel: Byte; ANDbits, XORbits: Pointer): HIcon;

This function creates an icon that has specified width, height, colors, and bit patterns.

Parameters

hInstance	**HANDLE** Identifies an instance of the module creating the icon.
nWidth	**int** Specifies the width in pixels of the icon.
nHeight	**int** Specifies the height in pixels of the icon.

nPlanes	**BYTE** Specifies the number of planes in the XOR mask of the icon.
nBitsPixel	**BYTE** Specifies the number of bits per pixel in the XOR mask of the icon.
lpANDbits	**LPSTR** Points to an array of bytes that contains the bit values for the AND mask of the icon. This array must specify a monochrome mask.
lpXORbits	**LPSTR** Points to an array of bytes that contains the bit values for the XOR mask of the icon. This can be the bits of a monochrome or device-dependent color bitmap.

Return value The return value identifies an icon if the function is successful. Otherwise, it is NULL.

CreateMenu

Syntax HMENU CreateMenu( )
function CreateMenu: HMenu;

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

Parameters None.

Return value The return value identifies the newly created menu. It is NULL if the menu cannot be created.

CreateMetaFile

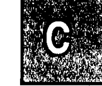
Syntax HANDLE CreateMetaFile(lpFilename)
function CreateMetaFile(FileName: PChar): THandle;

This function creates a metafile device context.

Parameters

lpFilename	**LPSTR** Points to a null-terminated character string that specifies the name of the metafile. If the *lpFilename* parameter is NULL, a device context for a memory metafile is returned.

Return value The return value identifies a metafile device context if the function is successful. Otherwise, it is NULL.



CreatePalette 3.0

Syntax HPALETTE CreatePalette(lpLogPalette)
function CreatePalette(var LogPalette: TLogPalette): HPalette;

This function creates a logical color palette.

Parameters *lpLogPalette* **LPLOGPALETTE** Points to a **LOGPALETTE** data structure that contains information about the colors in the logical palette.

Return value The return value identifies a logical palette if the function was successful. Otherwise, it is NULL.

CreatePatternBrush

Syntax HBRUSH CreatePatternBrush(hBitmap)
function CreatePatternBrush(Bitmap: HBitmap): HBrush;

This function creates a logical brush that has the pattern specified by the *hBitmap* parameter. The brush can subsequently be selected for any device that supports raster operations. For more information, see the RC_BITBLT raster capability in the **GetDeviceCaps** function, later in this chapter.

Parameters *hBitmap* **HBITMAP** Identifies the bitmap. It is assumed to have been created by using the **CreateBitmap**, **CreateBitmapIndirect**, **LoadBitmap**, or **CreateCompatibleBitmap** function. The minimum size for a bitmap to be used in a fill pattern is 8-by-8.

Return value The return value identifies a logical brush if the function is successful. Otherwise, it is NULL.

Comments A pattern brush can be deleted without affecting the associated bitmap by using the **DeleteObject** function. This means the bitmap can be used to create any number of pattern brushes.

A brush created using a monochrome (one plane, one bit per pixel) bitmap is drawn using the current text and background colors. Pixels represented by a bit set to 0 will be drawn with the current text color, and pixels represented by a bit set to 1 will be drawn with the current background color.

CreatePen

Syntax HPEN CreatePen(nPenStyle, nWidth, crColor)
function CreatePen(PenStyle, Width: Integer; Color: TColorRef): HPen;

This function creates a logical pen having the specified style, width, and color. The pen can be subsequently selected as the current pen for any device.

Parameters

nPenStyle **int** Specifies the pen style. It can be any one of the following values:

Pen Style	Value
PS_SOLID	0
PS_DASH	1
PS_DOT	2
PS_DASHDOT	3
PS_DASHDOTDOT	4
PS_NULL	5
PS_INSIDEFRAME	6

If the width of the pen is greater than 1 and the pen style is PS_INSIDEFRAME, the line is drawn inside the frame of all primitives except polygons and polylines; the pen is drawn with a logical (dithered) color if the pen color does not match an available RGB value. The PS_INSIDEFRAME style is identical to PS_SOLID if the pen width is less than or equal to 1

nWidth **int** Specifies the width of the pen (in logical units).

crColor **COLORREF** Specifies the color of the pen.

Return value The return value identifies a logical pen if the function is successful. Otherwise, it is NULL.

Comments Pens with a physical width greater than one pixel will always have either null or solid style or will be dithered if the pen style is PS_INSIDEFRAME.

CreatePenIndirect

Syntax HPEN CreatePenIndirect(lpLogPen)
function CreatePenIndirect(var LogPen: TLogPen): HPen;

This function creates a logical pen that has the style, width, and color given in the data structure pointed to by the *lpLogPen* parameter.

Parameters *lpLogPen* — **LOGPEN FAR *** Points to the **LOGPEN** data structure that contains information about the logical pen.

Return value The return value identifies a logical pen object if the function is successful. Otherwise, it is NULL.

Comments Pens with a physical width greater than one pixel will always have either null or solid style or will be dithered if the pen style is PS_INSIDEFRAME.

CreatePolygonRgn

Syntax HRGN CreatePolygonRgn(lpPoints, nCount, nPolyFillMode)
function CreatePolygonRgn(var Points; Count, PolyFillMode: Integer): HRgn;

This function creates a polygonal region.

Parameters

lpPoints	**LPPOINT** Points to an array of **POINT** data structures. Each point specifies the *x*- and *y*-coordinates of one vertex of the polygon.
nCount	**int** Specifies the number of points in the array.
nPolyFillMode	**int** Specifies the polygon-filling mode to be used for filling the region. It can be ALTERNATE or WINDING (for an explanation of these modes, see the **SetPolyFillMode** function, later in this chapter).

Return value The return value identifies a new region if the function is successful. Otherwise, it is NULL.

CreatePolyPolygonRgn 3.0

Syntax HRGN CreatePolyPolygonRgn(lpPoints, lpPolyCounts, nCount, nPolyFillMode)
function CreatePolyPolygonRgn(var Points; var PolyCounts; Count, PolyFillMode: Integer): HRgn;

This function creates a region consisting of a series of closed polygons. The region is filled using the mode specified by the *nPolyFillMode* parameter. The polygons may overlap, but they do not have to overlap.

Parameters	*lpPoints*	**LPPOINT** Points to an array of **POINT** data structures that define the vertices of the polygons. Each polygon must be a closed polygon. The polygons are not automatically closed. The polygons are specified consecutively.
	lpPolyCounts	**LPINT** Points to an array of integers, each of which specifies the number of points in one of the polygons in the *lpPoints* array.
	nCount	**int** Specifies the total number of integers in the *lpPolyCounts* array.
	nPolyFillMode	**int** Specifies the filling mode for the region. The *nPolyFillMode* parameter may be either of the following values:

Value	Meaning
ALTERNATE	Selects alternate mode.
WINDING	Selects winding number mode.

Return value The return value identifies the region if the function was successfull. Otherwise, it is NULL.

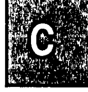
Comments In general, the polygon fill modes differ only in cases where a complex, overlapping polygon must be filled (for example, a five-sided polygon that forms a five-pointed star with a pentagon in the center). In such cases, ALTERNATE mode fills every other enclosed region within the polygon (that is, the points of the star), but WINDING mode fills all regions (that is, the points and the pentagon).

When the filling mode is ALTERNATE, GDI fills the area between odd-numbered and even-numbered polygon sides on each scan line. That is, GDI fills the area between the first and second side, between the third and fourth side, and so on.

To fill all parts of the region, WINDING mode causes GDI to compute and draw a border that encloses the region but does not overlap. For example, in WINDING mode, the five-sided polygon that forms the star is computed as a ten-sided polygon with no overlapping sides; the resulting star is filled.

CreatePopupMenu 3.0

Syntax HMENU CreatePopupMenu( )
function CreatePopupMenu: HMenu;



This function creates and returns a handle to an empty pop-up menu. An application adds items to the pop-up menu by calling **InsertMenu** and **AppendMenu**. The application can add the pop-up menu to an existing menu or pop-up menu, or it may display and track selections on the pop-up menu by calling **TrackPopupMenu**.

Parameters None.

Return value The return value identifies the newly created menu. It is NULL if the menu cannot be created.

CreateRectRgn

Syntax HRGNCreateRectRgn(X1, Y1, X2, Y2)
function CreateRectRgn(X1, Y1, X2, Y2: Integer): HRgn;

This function creates a rectangular region.

Parameters

X1	**int** Specifies the *x*-coordinate of the upper-left corner of the region.
Y1	**int** Specifies the *y*-coordinate of the upper-left corner of the region.
X2	**int** Specifies the *x*-coordinate of the lower-right corner of the region.
Y2	**int** Specifies the *y*-coordinate of the lower-right corner of the region.

Return value The return value identifies a new region if the function is successful. Otherwise, it is NULL.

Comments The width of the rectangle, specified by the absolute value of *X2* – *X1*, must not exceed 32,767 units. This limit applies to the height of the rectangle as well.

CreateRectRgnIndirect

Syntax HRGN CreateRectRgnIndirect(lpRect)
function CreateRectRgnIndirect(var Rect: TRect): HRgn;

This function creates a rectangular region.

Parameters *lpRect* **LPRECT** Points to a **RECT** data structure that contains the coordinates of the upper-left and lower-right corners of the region.

Return value The return value identifies a new region if the function is successful. Otherwise, it is NULL.

Comments The width of the rectangle must not exceed 32,767 units. This limit applies to the height of the rectangle as well.

CreateRoundRectRgn 3.0

Syntax HRGN CreateRoundRectRgn(X1, Y1, X2, Y2, X3, Y3)
function CreateRoundRectRgn(X1, Y1, X2, Y2, X3, Y3: Integer): HRgn;

This function creates a rectangular region with rounded corners.

Parameters

X1	**int** Specifies the *x*-coordinate of the upper-left corner of the region.
Y1	**int** Specifies the *y*-coordinate of the upper-left corner of the region.
X2	**int** Specifies the *x*-coordinate of the lower-right corner of the region.
Y2	**int** Specifies the *y*-coordinate of the lower-right corner of the region.
X3	**int** Specifies the width of the ellipse used to create the rounded corners.
Y3	**int** Specifies the height of the ellipse used to create the rounded corners.

Return value The return value identifies a new region if the function was successful. Otherwise, it is NULL.

Comments The width of the rectangle, specified by the absolute value of *X2* – *X1*, must not exceed 32,767 units. This limit applies to the height of the rectangle as well.

CreateSolidBrush

Syntax HBRUSH CreateSolidBrush(crColor)
function CreateSolidBrush(Color: TColorRef): HBrush;

This function creates a logical brush that has the specified solid color. The brush can subsequently be selected as the current brush for any device.

Parameters *crColor* **COLORREF** Specifies the color of the brush

Return value The return value identifies a logical brush if the function is successful. Otherwise, it is NULL.

CreateWindow

Syntax HWND CreateWindow(lpClassName, lpWindowName, dwStyle, X, Y, nWidth, nHeight, hWndParent, hMenu, hInstance, lpParam)
function CreateWindow(ClassName, WindowName: PChar; Style: Longint; X, Y, Width, Height: Integer; WndParent: HWnd; Menu: HMenu; Instance: THandle; Param: Pointer): HWnd;

This function creates an overlapped, pop-up, or child window. The **CreateWindow** function specifies the window class, window title, window style, and (optionally) initial position and size of the window. The **CreateWindow** function also specifies the window's parent (if any) and menu.

For overlapped, pop-up, and child windows, the **CreateWindow** function sends WM_CREATE, WM_GETMINMAXINFO, and WM_NCCREATE messages to the window. The *lParam* parameter of the WM_CREATE message contains a pointer to a **CREATESTRUCT** data structure. If WS_VISIBLE style is given, **CreateWindow** sends the window all the messages required to activate and show the window.

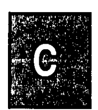
If the window style specifies a title bar, the window title pointed to by the *lpWindowName* parameter is displayed in the title bar. When using **CreateWindow** to create controls such as buttons, check boxes, and text controls, the *lpWindowName* parameter specifies the text of the control.

Parameters

lpClassName	**LPSTR** Points to a null-terminated character string that names the window class. The class name can be any name registered with the **RegisterClass** function or any of the predefined control-class names specified in Table 4.2, "Control classes.".
lpWindowName	**LPSTR** Points to a null-terminated character string that represents the window name.
dwStyle	**DWORD** Specifies the style of window being created. It can be any combination of the styles given in Table 4.3, "Window styles," the control styles given in Table 4.4,

"Control styles," or a combination of styles created by using the bitwise OR operator.

X	**int** Specifies the initial *x*-position of the window. For an overlapped or pop-up window, the *X* parameter is the initial *x*-coordinate of the window's upper-left corner (in screen coordinates). If this value is CW_USEDEFAULT, Windows selects the default position for the window's upper-left corner. For a child window, *X* is the *x*-coordinate of the upper-left corner of the window in the client area of its parent window.
Y	**int** Specifies the initial *y*-position of the window. For an overlapped window, the *Y* parameter is the initial *y*-coordinate of the window's upper-left corner. For a pop-up window, *Y* is the *y*-coordinate (in screen coordinates) of the upper-left corner of the pop-up window. For list-box controls, *Y* is the *y*-coordinate of the upper-left corner of the control's client area. For a child window, *Y* is the *y*-coordinate of the upper-left corner of the child window. All of these coordinates are for the window, not the window's client area.
nWidth	**int** Specifies the width (in device units) of the window. For overlapped windows, the *nWidth* parameter is either the window's width (in screen coordinates) or CW_USEDEFAULT. If *nWidth* is CW_USEDEFAULT, Windows selects a default width and height for the window (the default width extends from the initial *x*-position to the right edge of the screen, and the default height extends from the initial *y*-position to the top of the icon area).
nHeight	**int** Specifies the height (in device units) of the window. For overlapped windows, the *nHeight* parameter is the window's height in screen coordinates. If the *nWidth* parameter is CW_USEDEFAULT, Windows ignores *nHeight*.
hWndParent	**HWND** Identifies the parent or owner window of the window being created. A valid window handle must be supplied when creating a child window or an owned window. An owned window is an overlapped window that is destroyed when its owner window is destroyed, hidden when its owner is made iconic, and which is always displayed on top of its owner window. For pop-



	up windows, a handle can be supplied, but is not required. If the window does not have a parent or is not owned by another window, the **hWndParent** parameter must be set to NULL.
hMenu	**HMENU** Identifies a menu or a child-window identifier. The meaning depends on the window style. For overlapped or pop-up windows, the *hMenu* parameter identifies the menu to be used with the window. It can be NULL, if the class menu is to be used. For child windows, *hMenu* specifies the child-window identifier, an integer value that is used by a dialog-box control to notify its parent of events (such as the EN_HSCROLL message). The child-window identifier is determined by the application and should be unique for all child windows with the same parent window.
hInstance	**HANDLE** Identifies the instance of the module to be associated with the window.
lpParam	**LPSTR** Points to a value that is passed to the window through the **CREATESTRUCT** data structure referenced by the *lParam* parameter of the WM_CREATE message. If an application is calling **CreateWindow** to create a multiple document interface (MDI) client window, *lpParam* must point to a **CLIENTCREATESTRUCT** data structure.

Return value The return value identifies the new window. It is NULL if the window is not created.

Comments For overlapped windows where the *X* parameter is CW_USEDEFAULT, the *Y* parameter can be one of the show-style parameters described with the **ShowWindow** function, or, for the first overlapped window to be created by the application, it can be the *nCmdShow* parameter passed to the WinMain function.

Table 4.2 lists the window control classes; Table 4.3 lists the window styles; Table 4.4 lists the control styles:

Table 4.2 Control classes

Class	Meaning
BUTTON	Designates a small rectangular child window that represents a button the user can turn on or off by clicking it. Button controls can be used alone or in groups, and can either be labeled or appear without text. Button controls typically change appearance when the user clicks them.

Table 4.2: Control classes (continued)

COMBOBOX	Designates a control consisting of a selection field similar to an edit control plus a list box. The list box may be displayed at all times or may be dropped down when the user selects a "pop box" next to the selection field. Depending on the style of the combo box, the user can or cannot edit the contents of the selection field. If the list box is visible, typing characters into the selection box will cause the first list box entry that matches the characters typed to be highlighted. Conversely, selecting an item in the list box displays the selected text in the selection field.
EDIT	Designates a rectangular child window in which the user can enter text from the keyboard. The user selects the control, and gives it the input focus by clicking it or moving to it by using the TAB key. The user can enter text when the control displays a flashing caret. The mouse can be used to move the cursor and select characters to be replaced, or to position the cursor for inserting characters. The BACKSPACE key can be used to delete characters. Edit controls use the variable-pitch system font and display ANSI characters. Applications compiled to run with previous versions of Windows display text with a fixed-pitch system font unless they have been marked by the Windows 3.0 **MARK** utility with the **MEMORY FONT** option. An application can also send the WM_SETFONT message to the edit control to change the default font. Edit controls expand tab characters into as many space characters as are required to move the cursor to the next tab stop. Tab stops are assumed to be at every eighth character position.
LISTBOX	Designates a list of character strings. This control is used whenever an application needs to present a list of names, such as filenames, that the user can view and select. The user can select a string by pointing to it and clicking. When a string is selected, it is highlighted and a notification message is passed to the parent window. A vertical or horizontal scroll bar can be used with a list-box control to scroll lists that are too long for the control window. The list box automatically hides or shows the scroll bar as needed.
MDICLIENT	Designates an MDI client window. The MDI client window receives messages which control the MDI application's child windows. The recommended style bits are WS_CLIPCHILDREN and WS_CHILD. To create a scrollable MDI client window which allows the user to scroll MDI child windows into view, an application can also use the WS_HSCROLL and WS_VSCROLL styles.

Table 4.2: Control classes (continued)

SCROLLBAR	Designates a rectangle that contains a thumb and has direction arrows at both ends. The scroll bar sends a notification message to its parent window whenever the user clicks the control. The parent window is responsible for updating the thumb position, if necessary. Scroll-bar controls have the same appearance and function as scroll bars used in ordinary windows. Unlike scroll bars, scroll-bar controls can be positioned anywhere in a window and used whenever needed to provide scrolling input for a window. The scroll-bar class also includes size-box controls. A size-box control is a small rectangle that the user can expand to change the size of the window.
STATIC	Designates a simple text field, box, or rectangle that can be used to label, box, or separate other controls. Static controls take no input and provide no output.

Table 4.3 Window styles

Class	Meaning
DS_LOCALEDIT	Specifies that edit controls in the dialog box will use memory in the application's data segment. By default, all edit controls in dialog boxes use memory outside the application's data segment. This feature may be suppressed by adding the DS_LOCALEDIT flag to the STYLE command for the dialog box. If this flag is not used, EM_GETHANDLE and EM_SETHANDLE messages must not be used since the storage for the control is not in the application's data segment. This feature does not affect edit controls created outside of dialog boxes.
DS_MODALFRAME	Creates a dialog box with a modal dialog-box frame that can be combined with a title bar and System menu by specifying the WS_CAPTION and WS_SYSMENU styles.
DS_NOIDLEMSG	Suppresses WM_ENTERIDLE messages that Windows would otherwise send to the owner of the dialog box while the dialog box is displayed.
DS_SYSMODAL	Creates a system-modal dialog box.
WS_BORDER	Creates a window that has a border.
WS_CAPTION	Creates a window that has a title bar (implies the WS_BORDER style). This style cannot be used with the WS_DLGFRAME style.
WS_CHILD	Creates a child window. Cannot be used with the WS_POPUP style.
WS_CHILDWINDOW	Creates a child window that has the WS_CHILD style.

Table 4.3: Window styles (continued)

WS_CLIPCHILDREN	Excludes the area occupied by child windows when drawing within the parent window. Used when creating the parent window.
WS_CLIPSIBLINGS	Clips child windows relative to each other; that is, when a particular child window receives a paint message, the WS_CLIPSIBLINGS style clips all other overlapped child windows out of the region of the child window to be updated. (If WS_CLIPSIBLINGS is not given and child windows overlap, it is possible, when drawing within the client area of a child window, to draw within the client area of a neighboring child window.) For use with the WS_CHILD style only.
WS_DISABLED	Creates a window that is initially disabled.
WS_DLGFRAME	Creates a window with a double border but no title.
WS_GROUP	Specifies the first control of a group of controls in which the user can move from one control to the next by using the DIRECTION keys. All controls defined with the WS_GROUP style after the first control belong to the same group. The next control with the WS_GROUP style ends the style group and starts the next group (that is, one group ends where the next begins). Only dialog boxes use this style.
WS_HSCROLL	Creates a window that has a horizontal scroll bar.
WS_ICONIC	Creates a window that is initially iconic. For use with the WS_OVERLAPPED style only.
WS_MAXIMIZE	Creates a window of maximum size.
WS_MAXIMIZEBOX	Creates a window that has a maximize box.
WS_MINIMIZE	Creates a window of minimum size.
WS_MINIMIZEBOX	Creates a window that has a minimize box.
WS_OVERLAPPED	Creates an overlapped window. An overlapped window has a caption and a border.
WS_OVERLAPPEDWINDOW	Creates an overlapped window having the WS_OVERLAPPED, WS_CAPTION, WS_SYSMENU, WS_THICKFRAME, WS_MINIMIZEBOX, and WS_MAXIMIZEBOX styles.
WS_POPUP	Creates a pop-up window. Cannot be used with the WS_CHILD style.
WS_POPUPWINDOW	Creates a pop-up window that has the WS_BORDER, WS_POPUP, and WS_SYSMENU styles. The WS_CAPTION style must be combined with the WS_POPUPWINDOW style to make the system menu visible.

Table 4.3: Window styles (continued)

WS_SYSMENU	Creates a window that has a System-menu box in its title bar. Used only for windows with title bars.
WS_TABSTOP	Specifies one of any number of controls through which the user can move by using the TAB key. The TAB key moves the user to the next control specified by the WS_TABSTOP style. Only dialog boxes use this style.
WS_THICKFRAME	Creates a window with a thick frame that can be used to size the window.
WS_VISIBLE	Creates a window that is initially visible. This applies to overlapped and pop-up windows. For overlapped windows, the *Y* parameter is used as a **ShowWindow** function parameter.
WS_VSCROLL	Creates a window that has a vertical scroll bar.

Table 4.4
Control styles

Style	**Meaning**
BUTTON class	
BS_AUTOCHECKBOX	Identical to BS_CHECKBOX, except that the button automatically toggles its state whenever the user clicks it.
BS_AUTORADIOBUTTON	Identical to BS_RADIOBUTTON, except that the button is checked, the application is notified by BN_CLICKED, and the checkmarks are removed from all other radio buttons in the group.
BS_AUTO3STATE	Identical to BS_3STATE, except that the button automatically toggles its state when the user clicks it.
BS_CHECKBOX	Designates a small rectangular button that may be checked; its border is bold when the user clicks the button. Any text appears to the right of the button.
BS_DEFPUSHBUTTON	Designates a button with a bold border. This button represents the default user response. Any text is displayed within the button. Windows sends a message to the parent window when the user clicks the button.
BS_GROUPBOX	Designates a rectangle into which other buttons are grouped. Any text is displayed in the rectangle's upper-left corner.
BS_LEFTTEXT	Causes text to appear on the left side of the radio button or check-box button. Use this style with the BS_CHECKBOX, BS_RADIOBUTTON, or BS_3STATE styles.
BS_OWNERDRAW	Designates an owner-draw button. The parent window is notified when the button is clicked.

Table 4.4: Control styles (continued)

	Notification includes a request to paint, invert, and disable the button.
BS_PUSHBUTTON	Designates a button that contains the given text. The control sends a message to its parent window whenever the user clicks the button.
BS_RADIOBUTTON	Designates a small circular button that can be checked; its border is bold when the user clicks the button. Any text appears to the right of the button. Typically, two or more radio buttons are grouped together to represent mutually exclusive choices, so no more than one button in the group is checked at any time.
BS_3STATE	Identical to BS_CHECKBOX, except that a button can be grayed as well as checked. The grayed state typically is used to show that a check box has been disabled.
COMBOBOX class	
CBS_AUTOHSCROLL	Automatically scrolls the text in the edit control to the right when the user types a character at the end of the line. If this style is not set, only text which fits within the rectangular boundary is allowed.
CBS_DROPDOWN	Similar to CBS_SIMPLE, except that the list box is not displayed unless the user selects an icon next to the selection field.
CBS_DROPDOWNLIST	Similar to CBS_DROPDOWN, except that the edit control is replaced by a static text item which displays the current selection in the list box.
CBS_HASSTRINGS	An owner-draw combo box contains items consisting of strings. The combo box maintains the memory and pointers for the strings so the application can use the LB_GETTEXT message to retrieve the text for a particular item.
CBS_OEMCONVERT	Text entered in the combo box edit control is converted from the ANSI character set to the OEM character set and then back to ANSI. This ensures proper character conversion when the application calls the **AnsiToOem** function to convert an ANSI string in the combo box to OEM characters. This style is most useful for combo boxes that contain filenames and applies only to combo boxes created with the CBS_SIMPLE or CBS_DROPDOWN styles.

Table 4.4: Control styles (continued)

CBS_OWNERDRAWFIXED	The owner of the list box is responsible for drawing its contents; the items in the list box are all the same height.
CBS_OWNERDRAWVARIABLE	The owner of the list box is responsible for drawing its contents; the items in the list box are variable in height.
CBS_SIMPLE	The list box is displayed at all times. The current selection in the list box is displayed in the edit control.
CBS_SORT	Automatically sorts strings entered into the list box.
EDIT class	
ES_AUTOHSCROLL	Automatically scrolls text to the right by 10 characters when the user types a character at the end of the line. When the user presses the ENTER key, the control scrolls all text back to position zero.
ES_AUTOVSCROLL	Automatically scrolls text up one page when the user presses ENTER on the last line.
ES_CENTER	Centers text in a multiline edit control.
ES_LEFT	Aligns text flush-left.
ES_LOWERCASE	Converts all characters to lowercase as they are typed into the edit control.
ES_MULTILINE	Designates multiple-line edit control. (The default is single-line.) If the ES_AUTOVSCROLL style is specified, the edit control shows as many lines as possible and scrolls vertically when the user presses the ENTER key. If ES_AUTOVSCROLL is not given, the edit control shows as many lines as possible and beeps if ENTER is pressed when no more lines can be displayed. If the ES_AUTOHSCROLL style is specified, the multiple-line edit control automatically scrolls horizontally when the caret goes past the right edge of the control. To start a new line, the user must press ENTER. If ES_AUTOHSCROLL is not given, the control automatically wraps words to the beginning of the next line when necessary; a new line is also started if ENTER is pressed. The position of the wordwrap is determined by the window size. If the window size changes, the wordwrap position changes, and the text is redisplayed. Multiple-line edit controls can have scroll bars. An edit control with scroll bars processes its own scroll-bar messages. Edit controls without scroll bars scroll as described above,

Table 4.4: Control styles (continued)

	and process any scroll messages sent by the parent window.
ES_NOHIDESEL	Normally, an edit control hides the selection when the control loses the input focus, and inverts the selection when the control receives the input focus. Specifying ES_NOHIDESEL deletes this default action.
ES_OEMCONVERT	Text entered in the edit control is converted from the ANSI character set to the OEM character set and then back to ANSI. This ensures proper character conversion when the application calls the **AnsiToOem** function to convert an ANSI string in the edit control to OEM characters. This style is most useful for edit controls that contain filenames.
ES_PASSWORD	Displays all characters as an asterisk (*) as they are typed into the edit control. An application can use the EM_SETPASSWORDCHAR message to change the character that is displayed.
ES_RIGHT	Aligns text flush-right in a multiline edit control.
ES_UPPERCASE	Converts all characters to uppercase as they are typed into the edit control.
LISTBOX class	
LBS_EXTENDEDSEL	The user can select multiple items using the SHIFT key and the mouse or special key combinations.
LBS_HASSTRINGS	Specifies an owner-draw list box which contains items consisting of strings. The list box maintains the memory and pointers for the strings so the application can use the LB_GETTEXT message to retrieve the text for a particular item.
LBS_MULTICOLUMN	Specifies a multicolumn list box that is scrolled horizontally. The LB_SETCOLUMNWIDTH message sets the width of the columns.
LBS_MULTIPLESEL	String selection is toggled each time the user clicks or double-clicks the string. Any number of strings can be selected.
LBS_NOINTEGRALHEIGHT	The size of the list box is exactly the size specified by the application when it created the list box. Normally, Windows sizes a list box so that the list box does not display partial items.
LBS_NOREDRAW	List-box display is not updated when changes are made. This style can be changed at any

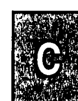


Table 4.4: Control styles (continued)

	time by sending a WM_SETREDRAW message.
LBS_NOTIFY	Parent window receives an input message whenever the user clicks or double-clicks a string.
LBS_OWNERDRAWFIXED	The owner of the list box is responsible for drawing its contents; the items in the list box are the same height.
LBS_OWNERDRAWVARIABLE	The owner of the list box is responsible for drawing its contents; the items in the list box are variable in height.
LBS_SORT	Strings in the list box are sorted alphabetically.
LBS_STANDARD	Strings in the list box are sorted alphabetically and the parent window receives an input message whenever the user clicks or double-clicks a string. The list box contains borders on all sides.
LBS_USETABSTOPS	Allows a list box to recognize and expand tab characters when drawing its strings. The default tab positions are 32 dialog units. (A dialog unit is a horizontal or vertical distance. One horizontal dialog unit is equal to 1/4 of the current dialog base width unit. The dialog base units are computed based on the height and width of the current system font. The **GetDialogBaseUnits** function returns the current dialog base units in pixels.)
LBS_WANTKEYBOARDINPUT	The owner of the list box receives WM_VKEYTOITEM or WM_CHARTOITEM messages whenever the user presses a key when the list box has input focus. This allows an application to perform special processing on the keyboard input.
SCROLLBAR class	
SBS_BOTTOMALIGN	Used with the SBS_HORZ style. The bottom edge of the scroll bar is aligned with the bottom edge of the rectangle specified by the *X*, *Y*, *nWidth*, and *nHeight* parameters given in the **CreateWindow** function. The scroll bar has the default height for system scroll bars.
SBS_HORZ	Designates a horizontal scroll bar. If neither the SBS_BOTTOMALIGN nor SBS_TOPALIGN style is specified, the scroll bar has the height, width, and position given in the **CreateWindow** function.
SBS_LEFTALIGN	Used with the SBS_VERT style. The left edge of the scroll bar is aligned with the left edge of the rectangle specified by the *X*, *Y*, *nWidth*, and *nHeight* parameters given in the

Table 4.4: Control styles (continued)

	CreateWindow function. The scroll bar has the default width for system scroll bars.
SBS_RIGHTALIGN	Used with the SBS_VERT style. The right edge of the scroll bar is aligned with the right edge of the rectangle specified by the *X*, *Y*, *nWidth*, and *nHeight* parameters given in the **CreateWindow** function. The scroll bar has the default width for system scroll bars.
SBS_SIZEBOX	Designates a size box. If neither the SBS_SIZEBOXBOTTOMRIGHTALIGN nor SBS_SIZEBOXTOPLEFTALIGN style is specified, the size box has the height, width, and position given in the **CreateWindow** function.
SBS_SIZEBOXBOTTOMRIGHTALIGN	Used with the SBS_SIZEBOX style. The lower-right corner of the size box is aligned with the lower-right corner of the rectangle specified by the *X*, *Y*, *nWidth*, and *nHeight* parameters given in the **CreateWindow** function. The size box has the default size for system size boxes.
SBS_SIZEBOXTOPLEFTALIGN	Used with the SBS_SIZEBOX style. The upper-left corner of the size box is aligned with the upper-left corner of the rectangle specified by the *X*, *Y*, *nWidth*, and *nHeight* parameters given in the **CreateWindow** function. The size box has the default size for system size boxes.
SBS_TOPALIGN	Used with the SBS_HORZ style. The top edge of the scroll bar is aligned with the top edge of the rectangle specified by the *X*, *Y*, *nWidth*, and *nHeight* parameters given in the **CreateWindow** function. The scroll bar has the default height for system scroll bars.
SBS_VERT	Designates a vertical scroll bar. If neither the SBS_RIGHTALIGN nor SBS_LEFTALIGN style is specified, the scroll bar has the height, width, and position given in the **CreateWindow** function.
STATIC class	
SS_BLACKFRAME	Specifies a box with a frame drawn with the same color as window frames. This color is black in the default Windows color scheme.
SS_BLACKRECT	Specifies a rectangle filled with the color used to draw window frames. This color is black in the default Windows color scheme.
SS_CENTER	Designates a simple rectangle and displays the given text centered in the rectangle. The text is

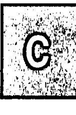


Table 4.4: Control styles (continued)

	formatted before it is displayed. Words that would extend past the end of a line are automatically wrapped to the beginning of the next centered line.
SS_GRAYFRAME	Specifies a box with a frame drawn with the same color as the screen background (desktop). This color is gray in the default Windows color scheme.
SS_GRAYRECT	Specifies a rectangle filled with the color used to fill the screen background. This color is gray in the default Windows color scheme.
SS_ICON	Designates an icon displayed in the dialog box. The given text is the name of an icon (not a filename) defined elsewhere in the resource file. The *nWidth* and *nHeight* parameters are ignored; the icon automatically sizes itself.
SS_LEFT	Designates a simple rectangle and displays the given text flush-left in the rectangle. The text is formatted before it is displayed. Words that would extend past the end of a line are automatically wrapped to the beginning of the next flush-left line.
SS_LEFTNOWORDWRAP	Designates a simple rectangle and displays the given text flush-left in the rectangle. Tabs are expanded, but words are not wrapped. Text that extends past the end of a line is clipped.
SS_NOPREFIX	Unless this style is specified, windows will interpret any "&" characters in the control's text to be accelerator prefix characters. In this case, the "&" is removed and the next character in the string is underlined. If a static control is to contain text where this feature is not wanted, SS_NOPREFIX may be added. This static-control style may be included with any of the defined static controls. You can combine SS_NOPREFIX with other styles by using the bitwise OR operator. This is most often used when filenames or other strings that may contain an "&" need to be displayed in a static control in a dialog box.
SS_RIGHT	Designates a simple rectangle and displays the given text flush-right in the rectangle. The text is formatted before it is displayed. Words that would extend past the end of a line are automatically wrapped to the beginning of the next flush-right line.
SS_SIMPLE	Designates a simple rectangle and displays a single line of text flush-left in the rectangle. The line of text cannot be shortened or altered in any way. (The control's parent window or

Table 4.4: Control styles (continued)

	dialog box must not process the WM_CTLCOLOR message.)
SS_USERITEM	Specifies a user-defined item.
SS_WHITEFRAME	Specifies a box with a frame drawn with the same color as window backgrounds. This color is white in the default Windows color scheme.
SS_WHITERECT	Specifies a rectangle filled with the color used to fill window backgrounds. This color is white in the default Windows color scheme.

CreateWindowEx 3.0

Syntax HWND CreateWindowEx(dwExStyle, lpClassName, lpWindowName, dwStyle, X, Y, nWidth, nHeight, hWndParent, hMenu, hInstance, lpParam)
function CreateWindowEx(ExStyle: Longint; ClassName, WindowName: PChar; Style: Longint; X, Y, Width, Height: Integer; WndParent: HWnd; Menu: HMenu; Instance: THandle; Param: Pointer): HWnd;

This function creates an overlapped, pop-up, or child window with an extended style specified in the dwExStyle parameter. Otherwise, this function is identical to the **CreateWindow** function. See the description of the **CreateWindow** function for more information on creating a window and for a full descriptions of the other parameters of **CreateWindowEx**.

Parameters

dwExStyle	**DWORD** Specifies the extended style of the window being created. Table 4.5, "Extended window styles," lists the extended window styles.
lpClassName	**LPSTR** Points to a null-terminated character string that names the window class.
lpWindowName	**LPSTR** Points to a null-terminated character string that represents the window name.
dwStyle	**DWORD** Specifies the style of window being created.
X	**int** Specifies the initial *x*-position of the window.
Y	**int** Specifies the initial *y*-position of the window.
nWidth	**int** Specifies the width (in device units) of the window.
nHeight	**int** Specifies the height (in device units) of the window.

C

hWndParent	**HWND** Identifies the parent or owner window of the window being created.
hMenu	**HMENU** Identifies a menu or a child-window identifier. The meaning depends on the window style.
hInstance	**HANDLE** Identifies the instance of the module to be associated with the window.
lpParam	**LPSTR** Points to a value that is passed to the window through the **CREATESTRUCT** data structure referenced by the *lParam* parameter of the WM_CREATE message.

Return value The return value identifies the new window. It is NULL if the window is not created.

Comments Table 4.5 lists the extended window styles.

Table 4.5 Extended window styles

Style	Meaning
WS_EX_DLGMODALFRAME	Designates a window with a double border that may optionally be created with a title bar by specifying the WS_CAPTION style flag in the *dwStyle* parameter.
WS_EX_NOPARENTNOTIFY	Specifies that a child window created with this style will not send the WM_PARENTNOTIFY message to its parent window when the child window is created or destroyed.
WS_EX_TOPMOST	Specifies that the window is a topmost window. A topmost window is always ordered above windows without this style, even when the topmost inactive. The **SetWindowPos** function enables and disables this feature.
function	Used to control topmost window style.

Table 4.2, "Control classes," lists the window control classes. Table 4.3, "Window styles," lists the window styles. Table 4.4, "Control styles," lists the control styles. See the description of the **CreateWindow** function for these tables.

DebugBreak 3.0

Syntax

```
void DebugBreak( )
procedure DebugBreak;
```

This function forces a break to the debugger.

Parameters None.

Return value None.

DefDlgProc 3.0

Syntax LONG DefDlgProc(hDlg, wMsg, wParam, lParam)
function DefDlgProc(Dlg: HWnd; Msg, wParam: Word; lParam: Longint): Longint;

This function provides default processing for any Windows messages that a dialog box with a private window class does not process.
All window messages that are not explicitly processed by the window function must be passed to the **DefDlgProc** function, not the **DefWindowProc** function. This ensures that all messages not handled by their private window procedure will be handled properly.

Parameters

hDlg	**HWND** Identifies the dialog box.
wMsg	**WORD** Specifies the message number.
wParam	**WORD** Specifies 16 bits of additional message-dependent information.
lParam	**DWORD** Specifies 32 bits of additional message-dependent information.

Return value The return value specifies the result of the message processing and depends on the actual message sent.

Comments The source code for the **DefDlgProc** function is provided on the SDK disks.

An application creates a dialog box by calling one of the following functions:

Function	Description
CreateDialog	Creates a modeless dialog box.
CreateDialogIndirect	Creates a modeless dialog box.
CreateDialogIndirectParam	Creates a modeless dialog box and passes data to it when it is created.
CreateDialogParam	Creates a modeless dialog box and passes data to it when it is created.
DialogBox	Creates a modal dialog box.
DialogBoxIndirect	Creates a modal dialog box.
DialogBoxIndirectParam	Creates a modal dialog box and passes data to it when it is created.
DialogBoxParam	Creates a modal dialog box and passes data to it when it is created.

DeferWindowPos 3.0

Syntax HANDLE DeferWindowPos(hWinPosInfo, hWnd, hWndInsertAfter, x, y, cx, cy, wFlags)
function DeferWindowPos(WinPosInfo: THandle; Wnd, WndInsertAfter: HWnd; X, Y, cX, cY: Integer; Flags: Word): THandle;

This function updates the multiple window-position data structure identified by the *hWinPosInfo* parameter for the window identified by *hWnd* parameter and returns the handle of the updated structure. The **EndDeferWindowPos** function uses the information in this structure to change the position and size of a number of windows simultaneously. The **BeginDeferWindowPos** function creates the multiple window-position data structure used by this function.

The *x* and *y* parameters specify the new position of the window, and the *cx* and *cy* parameters specify the new size of the window.

Parameters

hWinPosInfo	**HANDLE** Identifies a multiple window-position data structure that contains size and position information for one or more windows. This structure is returned by the **BeginDeferWindowPos** function or the most recent call to the **DeferWindowPos** function.
hWnd	**HWND** Identifies the window for which update information is to be stored in the data structure.
hWndInsertAfter	**HWND** Identifies the window following which the window identified by the *hWnd* parameter is to be updated.
x	**int** Specifies the *x*-coordinate of the window's upper-left corner.
y	**int** Specifies the *y*-coordinate of the window's upper-left corner.
cx	**int** Specifies the window's new width.
cy	**int** Specifies the window's new height.
wFlags	**WORD** Specifies one of eight possible 16-bit values that affect the size and position of the window. It must be one of the following values:

Value	Meaning
SWP_DRAWFRAME	Draws a frame (defined in the window's class description) around the window.
SWP_HIDEWINDOW	Hides the window.
SWP_NOACTIVATE	Does not activate the window.
SWP_NOMOVE	Retains current position (ignores the *x* and *y* parameters).
SWP_NOREDRAW	Does not redraw changes.
SWP_NOSIZE	Retains current size (ignores the *cx* and *cy* parameters).
SWP_NOZORDER	Retains current ordering (ignores the *hWndInsertAfter* parameter).
SWP_SHOWWINDOW	Displays the window.

Return value The return value identifies the updated multiple window-position data structure. The handle returned by this function may differ from the handle passed to the function as the *hWinPosInfo* parameter. The new handle returned by this function should be passed to the next call to **DeferWindowPos** or the **EndDeferWindowPos** function.

The return value is NULL if insufficient system resources are available for the function to complete successfully.

Comments If the SWP_NOZORDER flag is not specified, Windows places the window identified by the *hWnd* parameter in the position following the window identified by the *hWndInsertAfter* parameter. If *hWndInsertAfter* is NULL, Windows places the window identified by *hWnd* at the top of the list. If *hWndInsertAfter* is set to 1, Windows places the window identified by *hWnd* at the bottom of the list.

If the SWP_SHOWWINDOW or the SWP_HIDEWINDOW flags are set, scrolling and moving cannot be done simultaneously.

All coordinates for child windows are relative to the upper-left corner of the parent window's client area.

DefFrameProc 3.0

Syntax LONG DefFrameProc(hWnd, hWndMDIClient, wMsg, wParam, lParam)
function DefFrameProc(Wnd, MDIClient: HWnd; Msg, wParam: Word; lParam: Longint): Longint;

This function provides default processing for any Windows messages that the window function of a multiple document interface (MDI) frame

window does not process. All window messages that are not explicitly processed by the window function must be passed to the **DefFrameProc** function, not the **DefWindowProc** function.

Parameters

hWnd	**HWND** Identifies the MDI frame window.
hWndMDIClient	**HWND** Identifies the MDI client window.
wMsg	**WORD** Specifies the message number.
wParam	**WORD** Specifies 16 bits of additional message-dependent information.
lParam	**DWORD** Specifies 32 bits of additional message-dependent information.

Return value The return value specifies the result of the message processing and depends on the actual message sent. If the *hWndMDIClient* parameter is NULL, the return value is the same as for the **DefWindowProc** function.

Comments Normally, when an application's window procedure does not handle a message, it passes the message to the **DefWindowProc** function, which processes the message. MDI applications use the **DefFrameProc** and **DefMDIChildProc** functions instead of **DefWindowProc** to provide default message processing. All messages that an application would normally pass to **DefWindowProc** (such as nonclient messages and WM_SETTEXT) should be passed to **DefFrameProc** instead. In addition to these, **DefFrameProc** also handles the following messages:

Message	Default Processing by DefFrameProc
WM_COMMAND	The frame window of an MDI application receives the WM_COMMAND message to activate a particular MDI child window. The window ID accompanying this message will be the ID of the MDI child window assigned by Windows, starting with the first ID specified by the application when it created the MDI client window. This value of the first ID must not conflict with menu-item IDs.
WM_MENUCHAR	When the ALT+HYPHEN key is pressed, the control menu of the active MDI child window will be selected.
WM_SETFOCUS	**DefFrameProc** passes focus on to the MDI client, which in turn passes the focus on to the active MDI child window.
WM_SIZE	If the frame window procedure passes this message to **DefFrameProc**, the MDI client window will be resized to fit in the new client area. If the frame window procedure sizes the MDI client to a different size, it should not pass the message to **DefWindowProc**.

DefHookProc

Syntax DWORDDefHookProc(code, wParam, lParam, lplpfnNextHook)
function DefHookProc(Code: Integer; wParam: Word; lParam: Longint; NextHook: TFarProc): Longint;

This function calls the next function in a chain of hook functions. A hook function is a function that processes events before they are sent to an application's message-processing loop in the WinMain function. When an application defines more than one hook function by using the **SetWindowsHook** function, Windows forms a linked list or hook chain. Windows places functions of the same type in a chain.

Parameters

code **int** Specifies a code used by the Windows hook function (also called the message filter function) to determine how to process the message.

wParam **WORD** Specifies the word parameter of the message that the hook function is processing.

lParam **DWORD** Specifies the long parameter of the message that the hook function is processing.

lplpfnNextHook **FARPROC FAR** * Points to a memory location that contains the **FARPROC** structure returned by the **SetWindowsHook** function. Windows changes the value at this location after an application calls the **UnhookWindowsHook** function.

Return value The return value specifies a value that is directly related to the *code* parameter.

DefineHandleTable 3.0

Syntax BOOL DefineHandleTable(wOffset)

function DefineHandleTable(Offset: Word): Bool;

This function creates a private handle table in an application's default data segment. The application stores in the table the segment addresses of global memory objects returned by the **GlobalLock** function. In real mode, Windows updates the corresponding address in the private handle table when it moves a global memory object. When Windows discards an object with a corresponding table entry, Windows replaces the address of the object in the table with the object's handle. Windows does not update

addresses in the private handle table in protected (standard or 386 enhanced) mode.

Parameters *wOffset* **WORD** Specifies the offset from the beginning of the data segment to the beginning of the private handle table. If *wOffset* is zero, Windows no longer updates the private handle table.

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

Comments The private handle table has the following format:

```
Count

Clear_Number

Entry[0]
:
Entry[Count-1]
```

The first **WORD** (*Count*) in the table specifies the number of entries in the table. The second **WORD** (*Clear_Number*) specifies the number of entries (from the beginning of the table) which Windows will set to zero when Windows updates its least-recently-used (LRU) memory list. The remainder of the table consists of an array of addresses returned by **GlobalLock**.

The application must initialize the *Count* field in the table before calling **DefineHandleTable**. The application can change either the *Count* or *Clearn_Number* fields at any time.

DefMDIChildProc 3.0

Syntax LONG DefMDIChildProc(hWnd, wMsg, wParam, lParam)
function DefMDIChildProc(Wnd: HWnd; Msg, wParam: Word; lParam: Longint): Longint;

This function provides default processing for any Windows messages that the window function of a multiple document interface (MDI) child window does not process. All window messages that are not explicitly processed by the window function must be passed to the **DefMDIChildProc** function, not the **DefWindowProc** function.

Parameters *hWnd* **HWND** Identifies the MDI child window.

wMsg **WORD** Specifies the message number.

wParam	**WORD** Specifies 16 bits of additional message-dependent information.
lParam	**DWORD** Specifies 32 bits of additional message-dependent information.

Return value The return value specifies the result of the message processing and depends on the actual message sent.

Comments This function assumes that the parent of the window identified by the *hWnd* parameter was created with the MDICLIENT class.

Normally, when an application's window procedure does not handle a message, it passes the message to the **DefWindowProc** function, which processes the message. MDI applications use the **DefFrameProc** and **DefMDIChildProc** functions instead of **DefWindowProc** to provide default message processing. All messages that an application would normally pass to **DefWindowProc** (such as nonclient messages and WM_SETTEXT) should be passed to **DefMDIChildProc** instead. In addition to these, **DefMDIChildProc** also handles the following messages:

Message	Default Processing by DefMDIChildProc
WM_CHILDACTIVATE	Performs activation processing when child windows are sized, moved, or shown. This message must be passed.
WM_GETMINMAXINFO	Calculates the size of a maximized MDI child window based on the current size of the MDI client window.
WM_MENUCHAR	Sends the key to the frame window.
WM_MOVE	Recalculates MDI client scroll bars, if they are present.
WM_SETFOCUS	Activates the child window if it is not the active MDI child.
WM_SIZE	Performs necessary operations when changing the size of a window, especially when maximizing or restoring an MDI child window. Failing to pass this message to **DefMDIChildProc** will produce highly undesirable results.
WM_SYSCOMMAND	Also handles the "next window" command.

DefWindowProc

Syntax LONGDefWindowProc(hWnd, wMsg, wParam, lParam)
function DefWindowProc(Wnd: HWnd; Msg, wParam: Word; lParam: Longint): Longint;

This function provides default processing for any Windows messages that a given application does not process. All window messages that are not explicitly processed by the class window function must be passed to the **DefWindowProc** function.

Parameters

hWnd	**HWND** Identifies the window that passes the message.
wMsg	**WORD** Specifies the message number.
wParam	**WORD** Specifies 16 bits of additional message-dependent information.
lParam	**DWORD** Specifies 32 bits of additional message-dependent information.

Return value The return value specifies the result of the message processing and depends on the actual message sent.

Comments The source code for the **DefWindowProc** function is provided on the SDK disks.

DeleteAtom

Syntax ATOM DeleteAtom(nAtom)
function DeleteAtom(AnAtom: TAtom): TAtom;

This function deletes an atom and, if the atom's reference count is zero, removes the associated string from the atom table.

An atom's reference count specifies the number of times the atom has been added to the atom table. The **AddAtom** function increases the count on each call; the **DeleteAtom** function decreases the count on each call. **DeleteAtom** removes the string only if the atom's reference count is zero.

Parameters

nAtom	**ATOM** Identifies the atom and character string to be deleted.

Return value The return value specifies the outcome of the function. It is NULL if the function is successful. It is equal to the *nAtom* parameter if the function failed and the atom has not been deleted.

DeleteDC

Syntax BOOL DeleteDC(hDC)
function DeleteDC(DC: HDC): Bool;

This function deletes the specified device context. If the *hDC* parameter is the last device context for a given device, the device is notified and all storage and system resources used by the device are released.

Parameters *hDC* **HDC** Identifies the device context.

Return value The return value specifies whether the device context is deleted. It is nonzero if the device context is successfully deleted (regardless of whether the deleted device context is the last context for the device). If an error occurs, the return value is zero.

Comments An application must not delete a device context whose handle was obtained by calling the **GetDC** function. Instead, it must call the **ReleaseDC** function to free the device context.

DeleteMenu 3.0

Syntax BOOL DeleteMenu(hMenu, nPosition, wFlags)
function DeleteMenu(Menu: HMenu; Position, Flags: Word): Bool;

This function deletes an item from the menu identified by the *hMenu* parameter; if the menu item has an associated pop-up menu, **DeleteMenu** destroys the handle by the pop-up menu and frees the memory used by the pop-up menu.

Parameters

hMenu **HMENU** Identifies the menu to be changed.

nPosition **WORD** Specifies the menu item which is to be deleted. If *wFlags* is set to MF_BYPOSITION, *nPosition* specifies the position of the menu item; the first item in the menu is at position 0. If *wFlags* is set to MF_BYCOMMAND, then *nPosition* specifies the command ID of the existing menu item.

wFlags **WORD** Specifies how the *nPosition* parameter is interpreted. It may be set to either MF_BYCOMMAND (the default) or MF_BYPOSITION.

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

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

DeleteMetaFile

Syntax BOOL DeleteMetaFile(hMF)
function DeleteMetaFile(MF: THandle): Bool;

This function deletes access to a metafile by freeing the system resources associated with that metafile. It does not destroy the metafile itself, but it invalidates the metafile handle, *hMF*. Access to the metafile can be reestablished by retrieving a new handle by using the **GetMetaFile** function.

Parameters *hMF* **HANDLE** Identifies the metafile to be deleted.

Return value The return value specifies whether the metafile handle is invalidated. It is nonzero if the metafile's system resources are deleted. It is zero if the *hMF* parameter is not a valid handle.

DeleteObject

Syntax BOOL DeleteObject(hObject)
function DeleteObject(Handle: THandle): Bool;

This function deletes a logical pen, brush, font, bitmap, region, or palette from memory by freeing all system storage associated with the object. After the object is deleted, the *hObject* handle is no longer valid.

Parameters *hObject* **HANDLE** Identifies a handle to a logical pen, brush, font, bitmap, region, or palette.

Return value The return value specifies whether the specified object is deleted. It is nonzero if the object is deleted. It is zero if the *hObject* parameter is not a valid handle or is currently selected into a device context.

Comments The object to be deleted should not be currently selected into a device context.

When a pattern brush is deleted, the bitmap associated with the brush is not deleted. The bitmap must be deleted independently.

An application must not delete a stock object.

DestroyCaret

Syntax void DestroyCaret( )
procedure DestroyCaret;

This function destroys the current caret shape, frees the caret from the window that currently owns it, and removes the caret from the screen if it is visible. The **DestroyCaret** function checks the ownership of the caret and destroys the caret only if a window in the current task owns it.

If the caret shape was previously a bitmap, **DestroyCaret** does not free the bitmap.

Parameters None.

Return value None.

Comments The caret is a shared resource. If a window has created a caret shape, it destroys that shape before it loses the input focus or becomes inactive.

DestroyCursor 3.0

Syntax BOOL DestroyCursor(hCursor)
function DestroyCursor(Cursor: HCursor): Bool;

This function destroys a cursor that was previously created by the **CreateCursor** function and frees any memory that the cursor occupied. It should not be used to destroy any cursor that was not created with the **CreateCursor** function.

Parameters

hCursor	**HCURSOR** Identifies the cursor to be destroyed. The cursor must not be in current use.

Return value The return value is nonzero if the function was successful. It is zero if the function failed.

DestroyIcon 3.0

Syntax BOOL DestroyIcon(hIcon)
function DestroyIcon(Icon: HIcon): Bool;

This function destroys an icon that was previously created by the **CreateIcon** function and frees any memory that the icon occupied. It

should not be used to destroy any icon that was not created with the **CreateIcon** function.

Parameters *hIcon* **HICON** Identifies the icon to be destroyed. The icon must not be in current use.

Return value The return value is nonzero if the function was successful. It is zero if the function failed.

DestroyMenu

Syntax BOOL DestroyMenu(hMenu)
function DestroyMenu(Menu: HMenu): Bool;

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

Parameters *hMenu* **HMENU** Identifies the menu to be destroyed.

Return value The return value specifies whether or not the specified menu is destroyed. It is nonzero if the menu is destroyed. Otherwise, it is zero.

DestroyWindow

Syntax BOOL DestroyWindow(hWnd)
function DestroyWindow(Wnd: HWnd): Bool;

This function destroys the specified window. The **DestroyWindow** function hides or permanently closes the window, sending the appropriate messages to the window to deactivate it and remove the input focus. It also destroys the window menu, flushes the application queue, destroys outstanding timers, removes clipboard ownership, and breaks the clipboard-viewer chain, if the window is at the top of the viewer chain. It sends WM_DESTROY and WM_NCDESTROY messages to the window.

If the given window is the parent of any windows, these child windows are automatically destroyed when the parent window is destroyed. **DestroyWindow** destroys child windows first, and then the window itself.

DestroyWindow also destroys modeless dialog boxes created by the **CreateDialog** function.

Parameters *hWnd* **HWND** Identifies the window to be destroyed.

Return value The return value specifies whether or not the specified window is destroyed. It is nonzero if the window is destroyed. Otherwise, it is zero.

DeviceCapabilities 3.0

Syntax DWORD DeviceCapabilities(lpDeviceName, lpPort, nIndex, lpOutput, lpDevMode)

type TDeviceCapabilities = function(DeviceName, Port:PChar; Index:Word, Output:PChar; var DevMode:TDevMode): Longint;

This function retrieves the capabilities of the printer device driver.

Parameters

lpDeviceName **LPSTR** Points to a null-terminated character string that contains the name of the printer device, such as "PCL/HP LaserJet."

lpPort **LPSTR** Points to a null-terminated character string that contains the name of the port to which the device is connected, such as LPT1:.

nIndex **WORD** Specifies the capabilities to query. It can be any one of the following values:

Value	Meaning
DC_BINNAMES	Copies a structure identical to that returned by the **ENUMPAPERBINS** escape. A printer driver does not need to support this index if it has only bins corresponding to predefined indexes, in which case no data is copied and the return value is 0. If the index is supported, the return value is the number of bins copied. If *lpOutput* is NULL, the return value is the number of bin entries required.
DC_BINS	Retrieves a list of available bins. The function copies the list to *lpOutput* as a **WORD** array. If *lpOutput* is NULL, the function returns the number of supported bins to allow the application the

	opportunity to allocate a buffer with the correct size. See the description of the **dmDefaultSource** field of the **DEVMODE** data structure for information on these values. An application can determine the name of device-specific bins by using the **ENUMPAPERBINS** escape.
DC_DRIVER	Returns the printer driver version number.
DC_DUPLEX	Returns the level of duplex support. The function returns 1 if the printer is capable of duplex printing. Otherwise, the return value is zero.
DC_EXTRA	Returns the number of bytes required for the device-specific portion of the **DEVMODE** data structure for the printer driver.
DC_FIELDS	Returns the **dmFields** field of the printer driver's **DEVMODE** data structure. The **dmFields** bitfield indicates which fields in the device-independent portion of the structure are supported by the printer driver.
DC_MAXEXTENT	Returns a **POINT** data structure containing the maximum paper size that the **dmPaperLength** and **dmPaperWidth** fields of the printer driver's **DEVMODE** data structure can specify.
DC_MINEXTENT	Returns a **POINT** data structure containing the minimum paper size that the **dmPaperLength** and **dmPaperWidth** fields of the printer driver's **DEVMODE** data structure can specify.
DC_PAPERS	Retrieves a list of supported paper sizes. The function copies the list to *lpOutput* as a **WORD**

		array and returns the number of entries in the array. If *lpOutput* is NULL, the function returns the number of supported paper sizes to allow the application the opportunity to allocate a buffer with the correct size. See the description of the **dmPaperSize** field of the **DEVMODE** data structure for information on these values.
	DC_PAPERSIZE	Copies the dimensions of supported paper sizes in tenths of a millimeter to an array of **POINT** structures in *lpOutput*. This allows an application to obtain information about nonstandard paper sizes.
	DC_SIZE	Returns the **dmSize** field of the printer driver's **DEVMODE** data structure.
	DC_VERSION	Returns the specification version to which the printer driver conforms.

lpOutput **LPSTR** Points to an array of bytes. The actual format of the array depends on the setting of *nIndex*. If set to zero, **DeviceCapabilities** returns the number of bytes required for the output data.

lpDevMode **DEVMODE FAR *** Points to a **DEVMODE** data structure. If *lpDevMode* is NULL, this function retrieves the current default initialization values for the specified printer driver. Otherwise, the function retrieves the values contained in the structure to which *lpDevMode* points.

Return value The return value depends on the setting of the *nIndex* parameter; see the description of that parameter for details.

Comments This function is supplied by the printer driver. An application must include the DRIVINIT.H file and call the **LoadLibrary** and **GetProcAddress** functions to call the **DeviceCapabilities** function.

DeviceMode

Syntax void DeviceMode(hWnd, hModule, lpDeviceName, lpOutput)
type TDeviceMode = procedure(Wnd:HWnd; Module:THandle; DeviceName, Output:PChar);

This function sets the current printing modes for the device identified by the *lpDestDevType* by prompting for those modes using a dialog box. An application calls the **DeviceMode** function to allow the user to change the printing modes of the corresponding device. The function copies the mode information to the environment block associated with the device and maintained by GDI.

Parameters

hWnd	**HWND** Identifies the window that will own the dialog box.
hModule	**HANDLE** Identifies the printer-driver module. The application should retrieve this handle by calling either the **GetModuleHandle** or **LoadLibrary** function.
lpDeviceName	**LPSTR** Points to a null-terminated character string that specifies the name of the specific device to be supported (for example, Epson FX-80). The device name is the same as the name passed to the **CreateDC** function.
lpOutput	**LPSTR** Points to a null-terminated character string that specifies the DOS file or device name for the physical output medium (file or output port). The output name is the same as the name passed to the **CreateDC** function.

Return value None.

Comments The **DeviceMode** function is actually part of the printer's device driver, and not part of GDI. To call this function, the application must load the printer device driver by calling **LoadLibrary** and retrieve the address of the function by using the **GetProcAddress** function. The application can then use the address to set up the printer.

DialogBox

Syntax int DialogBox(hInstance, lpTemplateName, hWndParent, lpDialogFunc)
function DialogBox(Instance: THandle; TemplateName: PChar; WndParent: HWnd; DialogFunc: TFarProc): Integer;

This function creates a modal dialog box that has the size, style, and controls specified by the dialog-box template given by the *lpTemplateName* parameter. The *hWndParent* parameter identifies the application window that owns the dialog box. The callback function pointed to by the *lpDialogFunc* parameter processes any messages received by the dialog box.

The **DialogBox** function does not return control until the callback function terminates the modal dialog box by calling the **EndDialog** function.

Parameters

hInstance	**HANDLE** Identifies an instance of the module whose executable file contains the dialog-box template.
lpTemplateName	**LPSTR** Points to a character string that names the dialog-box template. The string must be a null-terminated character string.
hWndParent	**HWND** Identifies the window that owns the dialog box.
lpDialogFunc	**FARPROC** Is the procedure-instance address of the dialog function. See the following "Comments" section for details.

Return value The return value specifies the value of the *nResult* parameter in the **EndDialog** function that is used to terminate the dialog box. Values returned by the application's dialog box are processed by Windows and are not returned to the application. The return value is –1 if the function could not create the dialog box.

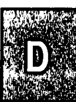
Comments The **DialogBox** function calls the **GetDC** function in order to obtain a display-context. Problems will result if all the display contexts in the Windows display-context cache have been retrieved by **GetDC** and **DialogBox** attempts to access another display context.

A dialog box can contain up to 255 controls. The callback function must use the Pascal calling convention and must be declared **FAR**.

Callback Function

int FAR PASCAL *DialogFunc(hDlg, wMsg, wParam, lParam)*
HWND *hDlg;*
WORD *wMsg;*
WORD *wParam;*
DWORD *lParam;*

DialogFunc is a placeholder for the application-supplied function name. The actual name must be exported by including it in an **EXPORTS** statement in the application's module-definition file.



Parameters	*hDlg*	Identifies the dialog box that receives the message.
	wMsg	Specifies the message number.
	wParam	Specifies 16 bits of additional message-dependent information.
	lParam	Specifies 32 bits of additional message-dependent information.

Return value The callback function should return nonzero if the function processes the message and zero if it does not.

Comments Although the callback function is similar to a window function, it must not call the **DefWindowProc** function to process unwanted messages. Unwanted messages are processed internally by the dialog-class window function.

The callback-function address, passed as the *lpDialogFunc* parameter, must be created by using the **MakeProcInstance** function.

DialogBoxIndirect

Syntax intDialogBoxIndirect(hInstance, hDialogTemplate, hWndParent, lpDialogFunc)
function DialogBoxIndirect(Instance, DialogTemplate: THandle; WndParent: HWnd; DialogFunc: TFarProc): Integer;

This function creates an application's modal dialog box that has the size, style, and controls specified by the dialog-box template associated with the *hDialogTemplate* parameter. The *hWndParent* parameter identifies the application window that owns the dialog box. The callback function pointed to by *lpDialogFunc* processes any messages received by the dialog box.
The **DialogBoxIndirect** function does not return control until the callback function terminates the modal dialog box by calling the **EndDialog** function.

Parameters	*hInstance*	**HANDLE** Identifies an instance of the module whose executable file contains the dialog-box template.
	hDialogTemplate	**HANDLE** Identifies a block of global memory that contains a **DLGTEMPLATE** data structure.
	hWndParent	**HWND** Identifies the window that owns the dialog box.

lpDialogFunc	**FARPROC** Is the procedure-instance address of the dialog function. See the following "Comments" section for details.

Return value The return value specifies the value of the *wResult* parameter specified in the **EndDialog** function that is used to terminate the dialog box. Values returned by the application's dialog box are processed by Windows and are not returned to the application. The return value is –1 if the function could not create the dialog box.

Comments A dialog box can contain up to 255 controls.
The callback function must use the Pascal calling convention and be declared **FAR**.

Callback Function

BOOL FAR PASCAL *DialogFunc*(*hDlg*, *wMsg*, *wParam*, *lParam*)
HWND *hDlg*;
WORD *wMsg*;
WORD *wParam*;
DWORD *lParam*;

DialogFunc is a placeholder for the application-supplied function name. The actual name must be exported by including it in an **EXPORTS** statement in the application's module-definition file.

Parameters

hDlg	Identifies the dialog box that receives the message.
wMsg	Specifies the message number.
wParam	Specifies 16 bits of additional message-dependent information.
lParam	Specifies 32 bits of additional message-dependent information.

Return value The callback function should return nonzero if the function processes the message and zero if it does not.

Comments Although the callback function is similar to a window function, it must not call the **DefWindowProc** function to process unwanted messages. Unwanted messages are processed internally by the dialog-class window function.

The callback-function address, passed as the *lpDialogFunc* parameter, must be created by using the **MakeProcInstance** function.

DialogBoxIndirectParam 3.0

Syntax int DialogBoxIndirectParam(hInstance, hDialogTemplate, hWndParent, lpDialogFunc, dwInitParam)
function DialogBoxIndirectParam(Instance, DialogTemplate: THandle; WndParent: HWnd; DialogFunc: TFarProc; InitParam: Longint): Integer;

This function creates an application's modal dialog box, sends a WM_INITDIALOG message to the dialog function before displaying the dialog box and passes *dwInitParam* as the message *lParam*. This message allows the dialog function to initialize the dialog-box controls.

For more information on creating an application modal dialog box, see the description of the **DialogBoxIndirect** function.

Parameters

hInstance	**HANDLE** Identifies an instance of the module whose executable file contains the dialog-box template.
hDialogTemplate	**HANDLE** Identifies a block of global memory that contains a **DLGTEMPLATE** data structure.
hWndParent	**HWND** Identifies the window that owns the dialog box.
lpDialogFunc	**FARPROC** Is the procedure-instance address of the dialog function. For details, see the "Comments" section in the description of the **DialogBoxIndirect** function.
dwInitParam	**DWORD** Is a 32-bit value which **DialogBoxIndirectParam** passes to the dialog function when it creates the dialog box.

Return value The return value specifies the value of the *wResult* parameter specified in the **EndDialog** function that is used to terminate the dialog box. Values returned by the application's dialog box are processed by Windows and are not returned to the application. The return value is –1 if the function could not create the dialog box.

DialogBoxParam 3.0

Syntax int DialogBoxParam(hInstance, lpTemplateName, hWndParent, lpDialogFunc, dwInitParam)
function DialogBoxParam(Instance: THandle; TemplateName: PChar; Wnd Parent: HWnd; DialogFunc: TFarProc; InitParam: Longint): Integer;

This function creates a modal dialog box, sends a WM_INITDIALOG message to the dialog function before displaying the dialog box, and

passes *dwInitParam* as the message *lParam*. This message allows the dialog function to initialize the dialog-box controls.

For more information on creating a modal dialog box, see the description of the **DialogBox** function.

Parameters

hInstance	**HANDLE** Identifies an instance of the module whose executable file contains the dialog-box template.
lpTemplateName	**LPSTR** Points to a character string that names the dialog-box template. The string must be a null-terminated character string.
hWndParent	**HWND** Identifies the window that owns the dialog box.
lpDialogFunc	**FARPROC** Is the procedure-instance address of the dialog function. For details, see the "Comments" section of the description of the **DialogBox** function.
dwInitParam	**DWORD** Is a 32-bit value which **DialogBoxParam** passes to the dialog function when it creates the dialog box.

Return value The return value specifies the value of the *nResult* parameter in the **EndDialog** function that is used to terminate the dialog box. Values returned by the application's dialog box are processed by Windows and are not returned to the application. The return value is –1 if the function could not create the dialog box.

DispatchMessage

Syntax LONG DispatchMessage(lpMsg)
function DispatchMessage(var Msg: TMsg): Longint;

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

Parameters

lpMsg	**LPMSG** Points to an **MSG** data structure that contains message information from the Windows application queue. The structure must contain valid message values. If *lpMsg* points to a WM_TIMER message and the *lParam* parameter of the WM_TIMER message is not NULL, then the *lParam* parameter is the address of a function that is called instead of the window function.

Return value The return value specifies the value returned by the window function. Its meaning depends on the message being dispatched, but generally the return value is ignored.

DlgDirList

Syntax int DlgDirList(hDlg, lpPathSpec, nIDListBox, nIDStaticPath, wFiletype)
function DlgDirList(Dlg: HWnd; PathSpec: PChar; IDListBox, IDStaticPath: Integer; Filetype: Word): Integer;

This function fills a list-box control with a file or directory listing. It fills the list box specified by the *nIDListBox* parameter with the names of all files matching the pathname given by the *lpPathSpec* parameter.

The **DlgDirList** function shows subdirectories enclosed in square brackets ([ ]), and shows drives in the form [–*x*–], where *x* is the drive letter.

The *lpPathSpec* parameter has the following form:

```
[[drive:]] [[ [[\]]directory[[\directory]]...\]] [[filename]]
```

In this example, *drive* is a drive letter, *directory* is a valid directory name, and *filename* is a valid filename that must contain at least one wildcard character. The wildcard characters are a question mark (?), meaning "match any character," and an asterisk (*), meaning "match any number of characters."

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

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

DlgDirList sends LB_RESETCONTENT and LB_DIR messages to the list box.

Parameters

hDlg	**HWND** Identifies the dialog box that contains the list box.
lpPathSpec	**LPSTR** Points to a pathname string. The string must be a null-terminated character string.
nIDListBox	**int** Specifies the identifier of a list-box control. If *nIDListBox* is zero, **DlgDirList** assumes that no list box exists and does not attempt to fill it.

nIDStaticPath — **int** Specifies the identifier of the static-text control used for displaying the current drive and directory. If *nIDStaticPath* is zero, **DlgDirList** assumes that no such text control is present.

wFiletype — **WORD** Specifies DOS file attributes of the files to be displayed. It can be any combination of the values given in Table 4.6, "DOS file attributes." Values can be combined by using the bitwise OR operator.

Return value The return value specifies the outcome of the function. It is nonzero if a listing was made, even an empty listing. A zero return value implies that the input string did not contain a valid search path.

The *wFiletype* parameter specifies the DOS attributes of the files to be listed. Table 4.6 describes these attributes.

Table 4.6
DOS file attributes

Attribute Value	Meaning
0x0000	Read/write data files with no additional attributes
0x0001	Read-only files
0x0002	Hidden files
0x0004	System files
0x0010	Subdirectories
0x0020	Archives
0x2000	LB_DIR flag[1]
0x4000	Drives
0x8000	Exclusive bit[2]

[1]If the LB_DIR flag is set, Windows places the messages generated by **DlgDirList** in the application's queue; otherwise they are sent directly to the dialog function.

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

DlgDirListComboBox 3.0

Syntax int DlgDirListComboBox(hDlg, lpPathSpec, nIDComboBox, nIDStaticPath, wFiletype)
function DlgDirListComboBox(Dlg: HWnd; PathSpec: PChar; IDComboBox, IDStaticPath: Integer; Filetype: Word): Integer;

This function fills the list box of a combo-box control with a file or directory listing. It fills the list box of the combo box specified by the *nIDComboBox* parameter with the names of all files matching the pathname given by the *lpPathSpec* parameter.

D

The **DlgDirListComboBox** function shows subdirectories enclosed in square brackets

([ ]), and shows drives in the form [–*x*–], where *x* is the drive letter.

The *lpPathSpec* parameter has the following form:

```
[[drive:]] [[ [[\]]directory[[\directory]]...\]] [[filename]]
```

In this example, *drive* is a drive letter, *directory* is a valid directory name, and *filename* is a valid filename that must contain at least one wildcard character. The wildcard characters are a question mark (?), meaning "match any character," and an asterisk (*), meaning "match any number of characters."

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

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

DlgDirListComboBox sends CB_RESETCONTENT and CB_DIR messages to the combo box.

Parameters

hDlg — **HWND** Identifies the dialog box that contains the combo box.

lpPathSpec — **LPSTR** Points to a pathname string. The string must be a null-terminated character string.

nIDComboBox — **int** Specifies the identifier of a combo-box control in a dialog box. If *nIDComboBox* is zero, **DlgDirListComboBox** assumes that no combo box exists and does not attempt to fill it.

nIDStaticPath — **int** Specifies the identifier of the static-text control used for displaying the current drive and directory. If *nIDStaticPath* is zero, **DlgDirListComboBox** assumes that no such text control is present.

wFiletype — **WORD** Specifies DOS file attributes of the files to be displayed. It can be any combination of the values given in Table 4.6, "DOS File Attributes." Refer to the description of the **DlgDirList** function for this table. Values can be combined by using the bitwise OR operator.

Return value The return value specifies the outcome of the function. It is nonzero if a listing was made, even an empty listing. A zero return value implies that the input string did not contain a valid search path.

DlgDirSelect

Syntax BOOL DlgDirSelect(hDlg, lpString, nIDListBox)
function DlgDirSelect(Dlg: HWnd; Str: PChar; IDListBox: Integer): Bool;

This function retrieves the current selection from a list box. It assumes that the list box has been filled by the **DlgDirList** function and that the selection is a drive letter, a file, or a directory name.

The **DlgDirSelect** function copies the selection to the buffer given by the *lpString* parameter. If the current selection is a directory name or drive letter, **DlgDirSelect** removes the enclosing square brackets (and hyphens, for drive letters) so that the name or letter is ready to be inserted into a new pathname. If there is no selection, *lpString* does not change.

DlgDirSelect sends LB_GETCURSEL and LB_GETTEXT messages to the list box.

Parameters

hDlg	**HWND** Identifies the dialog box that contains the list box.
lpString	**LPSTR** Points to a buffer that is to receive the selected pathname.
nIDListBox	**int** Specifies the integer ID of a list-box control in the dialog box.

Return value The return value specifies the status of the current list-box selection. It is nonzero if the current selection is a directory name. Otherwise, it is zero.

Comments The **DlgDirSelect** function does not allow more than one filename to be returned from a list box.

The list box must not be a multiple-selection list box. If it is, this function will not return a zero value and *lpString* will remain unchanged.

DlgDirSelectComboBox 3.0

Syntax BOOL DlgDirSelectComboBox(hDlg, lpString, nIDComboBox)
function DlgDirSelectComboBox(Dlg: HWnd; Str: PChar; IDComboBox: Integer): Bool;

D

This function retrieves the current selection from the list box of a combo box created with the CBS_SIMPLE style. It cannot be used with combo boxes created with either the CBS_DROPDOWN or CBS_DROPDOWNLIST style. It assumes that the list box has been filled by the **DlgDirListComboBox** function and that the selection is a drive letter, a file, or a directory name.

The **DlgDirSelectComboBox** function copies the selection to the buffer given by the *lpString* parameter. If the current selection is a directory name or drive letter, **DlgDirSelectComboBox** removes the enclosing square brackets (and hyphens, for drive letters) so that the name or letter is ready to be inserted into a new pathname. If there is no selection, *lpString* does not change.

DlgDirSelectComboBox sends CB_GETCURSEL and CB_GETLBTEXT messages to the combo box.

Parameters

hDlg	**HWND** Identifies the dialog box that contains the combo box.
lpString	**LPSTR** Points to a buffer that is to receive the selected pathname.
nIDComboBox	**int** Specifies the integer ID of the combo-box control in the dialog box.

Return value The return value specifies the status of the current combo-box selection. It is nonzero if the current selection is a directory name. Otherwise, it is zero.

Comments The **DlgDirSelectComboBox** function does not allow more than one filename to be returned from a combo box.

DOS3Call 3.0

procedure DOS3Call;

This function allows an application to issue a DOS function-request interrupt 21H. An application can use this function instead of a directly coded DOS 21H interrupt. The **DOS3Call** function executes somewhat faster than the equivalent DOS 21H software interrupt under Windows.

This function does not work properly when called from a discardable code segment while Windows is running in real mode. It does work properly in standard and 386 enhanced modes, and when called from a fixed code segment in real mode. An application can call the **GetWinFlags**

function to determine the mode in which Windows is running. An application must call INT 21H instead of **DOS3Call** if it is running in real mode from a discardable code segment. Otherwise the application must call **DOS3Call**.

An application can call this function only from an assembly-language routine. It is exported from KERNEL.EXE and is not defined in any Windows include files.

To use this function call, an application should declare it in an assembly-language program as shown:

```
extrn    DOS3Call    :far
```

If the application includes CMACROS.INC, the application declares it as shown:

```
extrnFP  Dos3Call
```

Before calling **DOS3Call**, all registers must be set as for an actual INT 21H. All registers at the function's exit are the same as for the corresponding INT 21H function.

Parameters None.

Return value The registers of the DOS function.

The following is an example of using **DOS3Call**:

```
extrn DOS3Call : far
:
; set registers
mov    ah, DOSFUNC
cCall DOS3Call
```

DPtoLP

Syntax BOOL DPtoLP(hDC, lpPoints, nCount)
function DPtoLP(DC: HDC; var Points; Count: Integer): Bool;

This function converts device points into logical points. The function maps the coordinates of each point specified by the *lpPoints* parameter from the device coordinate system into GDI's logical coordinate system. The conversion depends on the current mapping mode and the settings of the origins and extents for the device's window and viewport.

Parameters *hDC* **HDC** Identifies the device context.

lpPoints **LPPOINT** Points to an array of points. Each point must be a **POINT** data structure.

nCount **int** Specifies the number of points in the array.

Return value The return value specifies whether the conversion has taken place. It is nonzero if all points are converted. Otherwise, it is zero.

DrawFocusRect 3.0

Syntax void DrawFocusRect(hDC, lpRect)
procedure DrawFocusRect(DC: HDC; var Rect: TRect);

This function draws a rectangle in the style used to indicate focus.

Parameters *hDC* **HDC** Identifies the device context.

lpRect **LPRECT** Points to a **RECT** data structure that specifies the coordinates of the rectangle to be drawn.

Return value None.

Comments Since this is an XOR function, calling this function a second time with the same rectangle removes the rectangle from the display.

The rectangle drawn by this function cannot be scrolled. To scroll an area containing a rectangle drawn by this function, call **DrawFocusRect** to remove the rectangle from the display, scroll the area, and then call **DrawFocusRect** to draw the rectangle in the new position.

DrawIcon

Syntax BOOL DrawIcon(hDC, X, Y, hIcon)
function DrawIcon(DC: HDC; X, Y: Integer; Icon: HIcon): Bool;

This function draws an icon on the specified device. The **DrawIcon** function places the icon's upper-left corner at the location specified by the *X* and *Y* parameters. The location is subject to the current mapping mode of the device context.

Parameters *hDC* **HDC** Identifies the device context for a window.

X **int** Specifies the logical *x*-coordinate of the upper-left corner of the icon.

Y **int** Specifies the logical *y*-coordinate of the upper-left corner of the icon.

hIcon **HICON** Identifies the icon to be drawn.

Return value The return value specifies the outcome of the function. It is nonzero if the function is successful. Otherwise, it is zero.

Comments The icon resource must have been previously loaded by using the **LoadIcon** function. The MM_TEXT mapping mode must be selected prior to using this function.

DrawMenuBar

Syntax void DrawMenuBar(hWnd)
procedure DrawMenuBar(Wnd: HWnd);

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

Parameters *hWnd* **HWND** Identifies the window whose menu needs redrawing.

Return value None.

DrawText

Syntax int DrawText(hDC, lpString, nCount, lpRect, wFormat)
function DrawText(DC: HDC; Str: PChar; Count: Integer; var Rect: TRect; Format: Word): Integer;

This function draws formatted text in the rectangle specified by the *lpRect* parameter. It formats text by expanding tabs into appropriate spaces, justifying text to the left, right, or center of the given rectangle, and breaking text into lines that fit within the given rectangle. The type of formatting is specified by the *wFormat* parameter.

The **DrawText** function uses the device context's selected font, text color, and background color to draw the text. Unless the DT_NOCLIP format is used, **DrawText** clips the text so that the text does not appear outside the given rectangle. All formatting is assumed to have multiple lines unless the DT_SINGLELINE format is given.

Parameters *hDC* **HDC** Identifies the device context.

lpString **LPSTR** Points to the string to be drawn. If the *nCount* parameter is –1, the string must be null-terminated.

nCount	**int** Specifies the number of bytes in the string. If *nCount* is –1, then *lpString* is assumed to be a long pointer to a null-terminated string and **DrawText** computes the character count automatically.
lpRect	**LPRECT** Points to a **RECT** data structure that contains the rectangle (in logical coordinates) in which the text is to be formatted.
wFormat	**WORD** Specifies the method of formatting the text. It can be a combination of the values given in Table 4.7, "DrawText formats."

Return value The return value specifies the height of the text.

Comments If the selected font is too large for the specified rectangle, the **DrawText** function does not attempt to substitute a smaller font.

Table 4.7 lists the values for the *wFormat* parameter. These values can be combined by using the bitwise OR operator. Note that the DT_CALCRECT, DT_EXTERNALLEADING, DT_INTERNAL, DT_NOCLIP, and DT_NOPREFIX values cannot be used with the DT_TABSTOP value:

Table 4.7 DrawText formats

Value	Meaning
DT_BOTTOM	Specifies bottom-justified text. This value must be combined with DT_SINGLELINE.
DT_CALCRECT	Determines the width and height of the rectangle. If there are multiple lines of text, **DrawText** will use the width of the rectangle pointed to by the *lpRect* parameter and extend the base of the rectangle to bound the last line of text. If there is only one line of text, **DrawText** will modify the right side of the rectangle so that it bounds the last character in the line. In either case, **DrawText** returns the height of the formatted text but does not draw the text.
DT_CENTER	Centers text horizontally.
DT_EXPANDTABS	Expands tab characters. The default number of characters per tab is eight.
DT_EXTERNALLEADING	Includes the font external leading in line height. Normally, external leading is not included in the height of a line of text.
DT_LEFT	Aligns text flush-left.
DT_NOCLIP	Draws without clipping. **DrawText** is somewhat faster when DT_NOCLIP is used.
DT_NOPREFIX	Turns off processing of prefix characters. Normally, **DrawText** interprets the mnemonic-prefix character "&" as a directive to underscore the character that follows, and the mnemonic-prefix characters "&&"

Table 4.7: DrawText formats (continued)

	as a directive to print a single "&". By specifying DT_NOPREFIX, this processing is turned off.
DT_RIGHT	Aligns text flush-right.
DT_SINGLELINE	Specifies single line only. Carriage returns and linefeeds do not break the line.
DT_TABSTOP	Sets tab stops. The high-order byte of the *wFormat* parameter is the number of characters for each tab. The default number of characters per tab is eight.
DT_TOP	Specifies top-justified text (single line only).
DT_VCENTER	Specifies vertically centered text (single line only).
DT_WORDBREAK	Specifies word breaking. Lines are automatically broken between words if a word would extend past the edge of the rectangle specified by the *lpRect* parameter. A carriage return/line sequence will also break the line.

Ellipse

Syntax

BOOL Ellipse(hDC, X1, Y1, X2, Y2)
function Ellipse(DC: HDC; X1, Y1, X2, Y2: Integer): Bool;

This function draws an ellipse. The center of the ellipse is the center of the bounding rectangle specified by the *X1*, *Y1*, *X2*, and *Y2* parameters. The ellipse border is drawn with the current pen, and the interior is filled with the current brush.

If the bounding rectangle is empty, nothing is drawn.

Parameters

hDC	**HDC** Identifies the device context.
X1	**int** Specifies the logical *x*-coordinate of the upper-left corner of the bounding rectangle.
Y1	**int** Specifies the logical *y*-coordinate of the upper-left corner of the bounding rectangle.
X2	**int** Specifies the logical *x*-coordinate of the lower-right corner of the bounding rectangle.
Y2	**int** Specifies the logical *y*-coordinate of the lower-right corner of the bounding rectangle.

Return value

The return value specifies whether the ellipse is drawn. It is nonzero if the ellipse is drawn. Otherwise, it is zero.

Comments

The width of the rectangle, specified by the absolute value of *X2* – *X1*, must not exceed 32,767 units. This limit applies to the height of the rectangle as well.

The current position is neither used nor updated by this function.

EmptyClipboard

Syntax BOOL EmptyClipboard( )
function EmptyClipboard: Bool;

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

Parameters None.

Return value The return value specifies the status of the clipboard. It is nonzero if the clipboard is emptied. It is zero if an error occurs.

Comments The clipboard must be open when the **EmptyClipboard** function is called.

EnableHardwareInput

Syntax BOOL EnableHardwareInput(bEnableInput)
function EnableHardwareInput(EnableInput: Bool): Bool;

This function disables mouse and keyboard input. The input is saved if the *bEnableInput* parameter is TRUE and discarded if it is FALSE.

Parameters

bEnableInput	**BOOL** Specifies that the function should save input if the *bEnableInput* parameter is set to a nonzero value; specifies that the function should discard input if the *bEnableInput* parameter is set to zero.

Return value The return value specifies whether mouse and keyboard input is disabled.

It is nonzero if input was previously enabled. Otherwise, it is zero. The default return value is nonzero (TRUE).

Comments None.

EnableMenuItem

Syntax BOOL EnableMenuItem(hMenu, wIDEnableItem, wEnable)
function EnableMenuItem(Menu: HMenu; IDEnableItem, Enable: Word): Bool;

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

Parameters

hMenu **HMENU** Specifies the menu.

wIDEnableItem **WORD** Specifies the menu item to be checked. The *wIDEnableItem* parameter can specify pop-up menu items as well as menu items.

wEnable **WORD** Specifies the action to take. It can be a combination of MF_DISABLED, MF_ENABLED, or MF_GRAYED, with MF_BYCOMMAND or MF_BYPOSITION. These values can be combined by using the bitwise OR operator. These values have the following meanings:

Value	Meaning
MF_BYCOMMAND	Specifies that the *wIDEnableItem* parameter gives the menu item ID (MF_BYCOMMAND is the default ID).
MF_BYPOSITION	Specifies that the *wIDEnableItem* parameter gives the position of the menu item (the first item is at position zero).
MF_DISABLED	Menu item is disabled.
MF_ENABLED	Menu item is enabled.
MF_GRAYED	Menu item is grayed.

Return value The return value specifies the previous state of the menu item. The return value is –1 if the menu item does not exist.

Comments To disable or enable input to a menu bar, see the WM_SYSCOMMAND message.

EnableWindow

Syntax BOOL EnableWindow(hWnd, bEnable)
function EnableWindow(Wnd: HWnd; Enable: Bool): Bool;

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

The **EnableWindow** function enables mouse and keyboard input to a window if the *bEnable* parameter is nonzero, and disables it if *bEnable* is zero.

Parameters

hWnd	**HWND** Identifies the window to be enabled or disabled.
bEnable	**BOOL** Specifies whether the given window is to be enabled or disabled.

Return value The return value specifies the outcome of the function. It is nonzero if the window is enabled or disabled as specified. It is zero if an error occurs.

Comments A window must be enabled before it can be activated. For example, if an application is displaying a modeless dialog box and has disabled its main window, the main window must be enabled before the dialog box is destroyed. Otherwise, another window will get the input focus and be activated. If a child window is disabled, it is ignored when Windows tries to determine which window should get mouse messages.

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

EndDeferWindowPos 3.0

Syntax void EndDeferWindowPos(hWinPosInfo)
procedure EndDeferWindowPos(WinPosInfo: THandle);

This function simultaneously updates the position and size of one or more windows in a single screen-refresh cycle. The *hWinPosInfo* parameter identifies a multiple window-position data structure that contains the update information for the windows. The **Defer-WindowPos** function stores the update information in the data structure; the **BeginDefer-WindowPos** function creates the initial data structure used by these functions.

Parameters *hWinPosInfo* **HANDLE** Identifies a multiple window-position data structure that contains size and position information for one or more windows. This structure is returned by the **BeginDeferWindowPos** function or the most recent call to the **DeferWindowPos** function.

Return value None.

EndDialog

Syntax void EndDialog(hDlg, nResult)
procedure EndDialog(Dlg: HWnd; Result: Integer);

This function terminates a modal dialog box and returns the given result to the **DialogBox** function that created the dialog box. The **EndDialog** function is required to complete processing whenever the **DialogBox** function is used to create a modal dialog box. The function must be used in the dialog function of the modal dialog box and should not be used for any other purpose.

The dialog function can call **EndDialog** at any time, even during the processing of the WM_INITDIALOG message. If called during the WM_INITDIALOG message, the dialog box is terminated before it is shown or before the input focus is set.

EndDialog does not terminate the dialog box immediately. Instead, it sets a flag that directs the dialog box to terminate as soon as the dialog function ends. The **EndDialog** function returns to the dialog function, so the dialog function must return control to Windows.

Parameters

hDlg **HWND** Identifies the dialog box to be destroyed.

nResult **int** Specifies the value to be returned from the dialog box to the **DialogBox** function that created it.

Return value None.

EndPaint

Syntax void EndPaint(hWnd, lpPaint)
procedure EndPaint(Wnd: HWnd; var Paint: TPaintStruct);

This function marks the end of painting in the given window. The **EndPaint** function is required for each call to the **BeginPaint** function, but only after painting is complete.

Parameters

hWnd	**HWND** Identifies the window that is repainted.
lpPaint	**LPPAINTSTRUCT** Points to a **PAINTSTRUCT** data structure that contains the painting information retrieved by the **BeginPaint** function.

Return value None.

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

EnumChildWindows

Syntax BOOL EnumChildWindows(hWndParent, lpEnumFunc, lParam)
function EnumChildWindows(WndParent: HWnd; EnumFunc: TFarProc; lParam: Longint): Bool;

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

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

Parameters

hWndParent	**HWND** Identifies the parent window whose child windows are to be enumerated.
lpEnumFunc	**FARPROC** Is the procedure-instance address of the callback function.
lParam	**DWORD** Specifies the value to be passed to the callback function for the application's use.

Return value The return value specifies nonzero if all child windows have been enumerated. Otherwise, it is zero.

Comments This function does not enumerate pop-up windows that belong to the *hWndParent* parameter.

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

The callback function must use the Pascal calling convention and must be declared **FAR**.

Callback function

BOOL FAR PASCAL *EnumFunc(hWnd, lParam)*
HWND *hWnd;*
DWORD *lParam;*

EnumFunc is a placeholder for the application-supplied function name. The actual name must be exported by including it in an **EXPORTS** statement in the application's module-definition file.

Parameters

hWnd	Identifies the window handle.
lParam	Specifies the long parameter argument of the **EnumChildWindows** function.

Return value The callback function should return a nonzero value to continue enumeration; it should return zero to stop enumeration.

EnumClipboardFormats

Syntax

WORD EnumClipboardFormats(wFormat)
function EnumClipboardFormats(Format: Word): Word;

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

Parameters

wFormat	**WORD** Specifies a known format.

Return value The return value specifies the next known clipboard data format. It is zero if *wFormat* specifies the last format in the list of available formats. It is zero if the clipboard is not open.

Comments Before it enumerates the formats by using the **EnumClipboardFormats** function, an application must open the clipboard by using the **OpenClipboard** function.

The order that an application uses for putting alternative formats for the same data into the clipboard is the same order that the enumerator uses when returning them to the pasting application. The pasting application should use the first format enumerated that it can handle. This gives the donor a chance to recommend formats that involve the least loss of data.

EnumFonts

Syntax

int EnumFonts(hDC, lpFacename, lpFontFunc, lpData)
function EnumFonts(DC: HDC; FaceName: PChar; FontFunc: TFarProc; Data: Pointer): Integer;

This function enumerates the fonts available on a given device. For each font having the typeface name specified by the *lpFacename* parameter, the **EnumFonts** function retrieves information about that font and passes it to the function pointed to by the *lpFontFunc* parameter. The application-supplied callback function can process the font information as desired. Enumeration continues until there are no more fonts or the callback function returns zero.

Parameters

hDC	**HDC** Identifies the device context.
lpFacename	**LPSTR** Points to a null-terminated character string that specifies the typeface name of the desired fonts. If *lpFacename* is NULL, **EnumFonts** randomly selects and enumerates one font of each available typeface.
lpFontFunc	**FARPROC** Is the procedure-instance address of the callback function. See the following "Comments" section for details.
lpData	**LPSTR** Points to the application-supplied data. The data is passed to the callback function along with the font information.

Return value

The return value specifies the last value returned by the callback function. Its meaning is user-defined.

Comments

The address passed as the *lpFontFunc* parameter must be created by using the **MakeProcInstance** function.

The callback function must use the Pascal calling convention and must be declared **FAR**.

Callback function

int FAR PASCAL *FontFunc(lpLogFont, lpTextMetrics, nFontType, lpData)*
LPLOGFONT *lpLogFont;*
LPTEXTMETRICS *lpTextMetrics;*
short *nFontType;*
LPSTR *lpData;*

FontFunc is a placeholder for the application-supplied function name. The actual name must be exported by including it in an **EXPORTS** statement in the application's module-definition file.

Parameters

lpLogFont	Points to a **LOGFONT** data structure that contains information about the logical attributes of the font.
lpTextMetrics	Points to a **TEXTMETRIC** data structure that contains information about the physical attributes of the font.
nFontType	Specifies the type of the font.
lpData	Points to the application-supplied data passed by **EnumFonts**.

Return value The return value can be any integer.

Comments The AND (&) operator can be used with the RASTER_FONTTYPE and DEVICE_FONTTYPE constants to determine the font type. The RASTER_FONTTYPE bit of the *FontType* parameter specifies whether the font is a raster or vector font. If the bit is one, the font is a raster font; if zero, it is a vector font. The DEVICE_FONTTYPE bit of *FontType* specifies whether the font is a device- or GDI-based font. If the bit is one, the font is a device-based font; if zero, it is a GDI-based font.

If the device is capable of text transformations (scaling, italicizing, and so on) only the base font will be enumerated. The user must inquire into the device's text-transformation abilities to determine which additional fonts are available directly from the device. GDI can simulate the bold, italic, underlined, and strikeout attributes for any GDI-based font.

EnumFonts only enumerates fonts from the GDI internal table. This does not include fonts that are generated by a device, such as fonts that are transformations of fonts from the internal table. The **GetDeviceCaps** function can be used to determine which transformations a device can perform. This information is available by using the TEXTCAPS index.

GDI can scale GDI-based raster fonts by one to five horizontally and one to eight vertically, unless PROOF_QUALITY is being used.

EnumMetaFile

Syntax BOOL EnumMetaFile(hDC, hMF, lpCallbackFunc, lpClientData)
function EnumMetaFile(DC: HDC; MF: THandle; CallbackFunc: TFarProc; ClientData: Pointer): Bool;

This function enumerates the GDI calls within the metafile identified by the *hMF* parameter. The **EnumMetaFile** function retrieves each GDI call within the metafile and passes it to the function pointed to by the *lpCallbackFunc* parameter. This callback function, an application-supplied function, can process each GDI call as desired. Enumeration continues until there are no more GDI calls or the callback function returns zero.

Parameters

hDC	**HDC** Identifies the device context associated with the metafile.
hMF	**LOCALHANDLE** Identifies the metafile.
lpCallbackFunc	**FARPROC** Is the procedure-instance callback function. See the following "Comments" section for details.
lpClientData	**BYTE FAR** * Points to the callback-function data.

Return value The return value specifies the outcome of the function. It is nonzero if the callback function enumerates all the GDI calls in a metafile; otherwise, it returns zero.

Comments The callback function must use the Pascal calling convention and must be declared **FAR**.

Callback function

int FAR PASCAL *EnumFunc(hDC, lpHTable, lpMFR, nObj, lpClientData)*
HDC *hDC;*
LPHANDLETABLE *lpHTable;*
LPMETARECORD *lpMFR;*
int *nObj;*
BYTE FAR * *lpClientData;*

EnumFunc is a placeholder for the application-supplied function name. The actual name must be exported by including it in an **EXPORTS** statement in the application's module-definition file.

Parameters

hDC	Identifies the special device context that contains the metafile.
lpHTable	Points to a table of handles associated with the objects (pens, brushes, and so on) in the metafile.

lpMFR	Points to a metafile record contained in the metafile.
nObj	Specifies the number of objects with associated handles in the handle table.
lpClientData	Points to the application-supplied data.

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

EnumObjects

Syntax int EnumObjects(hDC, nObjectType, lpObjectFunc, lpData)
function EnumObjects(DC: HDC; ObjectType: Integer; ObjectFunc: TFarProc; Data: Pointer): Integer;

This function enumerates the pens and brushes available on a device. For each object that belongs to the given style, the callback function is called with the information for that object. The callback function is called until there are no more objects or the callback function returns zero.

Parameters

hDC	**HDC** Identifies the device context.
nObjectType	**int** Specifies the object type. It can be one of the following values: ■ OBJ_BRUSH ■ OBJ_PEN
lpObjectFunc	**FARPROC** Is the procedure-instance address of the application-supplied callback function. See the following "Comments" section for details.
lpData	**LPSTR** Points to the application-supplied data. The data is passed to the callback function along with the object information.

Return value The return value specifies the last value returned by the callback function. Its meaning is user-defined.

Comments The address passed as the *lpObjectFunc* parameter must be created by using the **MakeProcInstance** function.

The callback function must use the Pascal calling convention and must be declared **FAR**.

Callback function

int FAR PASCAL *ObjectFunc(lpLogObject, lpData)*
char FAR * *lpLogObject;*
char FAR * *lpData;*

ObjectFunc is a placeholder for the application-supplied function name. The actual name must be exported by including it in an **EXPORTS** statement in the application's module-definition file.

Parameters

lpLogObject	Points to a **LOGPEN** or **LOGBRUSH** data structure that contains information about the logical attributes of the object.
lpData	Points to the application-supplied data passed to the **EnumObjects** function.

EnumProps

Syntax

int EnumProps(hWnd, lpEnumFunc)
function EnumProps(Wnd: HWnd; EnumFunc: TFarProc): Integer;

This function enumerates all entries in the property list of the specified window. It enumerates the entries by passing them, one by one, to the callback function specified by *lpEnumFunc*. **EnumProps** continues until the last entry is enumerated or the callback function returns zero.

Parameters

hWnd	**HWND** Identifies the window whose property list is to be enumerated.
lpEnumFunc	**FARPROC** Is the procedure-instance address of the callback function. See the following "Comments" section for details.

Return value

The return value specifies the last value returned by the callback function. It is –1 if the function did not find a property for enumeration.

Comments

An application can remove only those properties which it has added. It should not remove properties added by other applications or by Windows itself.

The following restrictions apply to the callback function:

1. The callback function must not yield control or do anything that might yield control to other tasks.

2. The callback function can call the **RemoveProp** function. However, the **RemoveProp** function can remove only the property passed to the callback function through the callback function's parameters.
3. A callback function should not attempt to add properties.

The address passed in the *lpEnumFunc* parameter must be created by using the **MakeProcInstance** function.

Fixed data segments

The callback function must use the Pascal calling convention and must be declared **FAR**. In applications and dynamic libraries with fixed data segments and in dynamic libraries with moveable data segments that do not contain a stack, the callback function must have the form shown below.

Callback function

int FAR PASCAL *EnumFunc(hWnd, lpString, hData)*
HWND *hWnd;*
LPSTR *lpString;*
HANDLE *hData;*

EnumFunc is a placeholder for the application-supplied function name. The actual name must be exported by including it in an **EXPORTS** statement in the application's module-definition file.

Parameters

hWnd	Identifies a handle to the window that contains the property list.
lpString	Points to the null-terminated character string associated with the data handle when the application called the **SetProp** function to set the property. If the application passed an atom instead of a string to the **SetProp** function, the *lpString* parameter contains the atom in its low-order word, and the high-order word is zero.
hData	Identifies the data handle.

Return value

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

Moveable data segments

The callback function must use the Pascal calling convention and must be declared **FAR**. In applications with moveable data segments and in dynamic libraries whose moveable data segments also contain a stack, the callback function must have the form shown below.

Callback function

int FAR PASCAL *EnumFunc(hWnd, nDummy, pString, hData)*
HWND *hWnd;*
WORD *nDummy;*
PSTR *pString;*
HANDLE *hData;*

EnumFunc is a placeholder for the application-supplied function name. The actual name must be exported by including it in an **EXPORTS** statement in the application's module-definition file.

Parameters

hWnd	Identifies a handle to the window that contains the property list.
nDummy	Specifies a dummy parameter.
pString	Points to the null-terminated character string associated with the data handle when the application called the **SetProp** function to set the property. If the application passed an atom instead of a string to the **SetProp** function, the *pString* parameter contains the atom.
hData	Identifies the data handle.

Return value

The callback function can carry out any desired task. It should return a nonzero value to continue enumeration, or a zero value to stop it.

Comments

The alternate form above is required since movement of the data will invalidate any long pointer to a variable on the stack, such as the *lpString* parameter. The data segment typically moves if the callback function allocates more space in the local heap than is currently available.

EnumTaskWindows

Syntax BOOL EnumTaskWindows(hTask, lpEnumFunc, lParam)
function EnumTaskWindows(Task: THandle; EnumFunc: TFarProc; lParam: Longint): Bool;

This function enumerates all windows associated with the *hTask* parameter, which is returned by the **GetCurrentTask** function. (A task is any program that executes as an independent unit. All applications are executed as tasks and each instance of an application is a task.) The enumeration terminates when the callback function, pointed to by *lpEnumFunc*, returns FALSE.

Parameters

hTask	**HANDLE** Identifies the specified task. The **GetCurrentTask** function returns this handle.
lpEnumFunc	**FARPROC** Is the procedure-instance address of the window's callback function.
lParam	**DWORD** Specifies the 32-bit value that contains additional parameters that are sent to the callback function pointed to by *lpEnumFunc*.

Return value The return value specifies the outcome of the function. It is nonzero if all the windows associated with a particular task are enumerated. Otherwise, it is zero.

Comments The callback function must use the Pascal calling convention and must be declared **FAR**. The callback function must have the following form:

Callback function

BOOL FAR PASCAL *EnumFunc*(*hWnd*, *lParam*)
HWND *hWnd*;
DWORD *lParam*;

EnumFunc is a placeholder for the application-supplied function name. The actual name must be exported by including it in an **EXPORTS** statement in the application's module-definition file.

Parameters

hWnd	Identifies a window associated with the current task.
lParam	Specifies the same argument that was passed to the **EnumTaskWindows** function.

Return value The callback function can carry out any desired task. It must return a nonzero value to continue enumeration, or a zero value to stop it.

EnumWindows

Syntax BOOL EnumWindows(lpEnumFunc, lParam)
function EnumWindows(EnumFunc: TFarProc; lParam: Longint): Bool;

This function enumerates all parent windows on the screen by passing the handle of each window, in turn, to the callback function pointed to by the *lpEnumFunc* parameter. Child windows are not enumerated.

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

Parameters

lpEnumFunc	**FARPROC** Is the procedure-instance address of the callback function. See the following "Comments" section for details.
lParam	**DWORD** Specifies the value to be passed to the callback function for the application's use.

Return value The return value specifies the outcome of the function. It is nonzero if all windows have been enumerated. Otherwise, it is zero.

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

The callback function must use the Pascal calling convention and must be declared **FAR**. The callback function must have the following form:

Callback function

BOOL FAR PASCAL *EnumFunc(hWnd, lParam)*
HWND *hWnd;*
DWORD *lParam;*

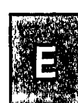
EnumFunc is a placeholder for the application-supplied function name. The actual name must be exported by including it in an **EXPORTS** statement in the application's module-definition file.

Parameters

hWnd	Identifies the window handle.
lParam	Specifies the 32-bit argument of the **EnumWindows** function.

Return value The function must return a nonzero value to continue enumeration, or zero to stop it.

EqualRect



Syntax BOOL EqualRect(lpRect1, lpRect2)
function EqualRect(var Rect1, Rect2: TRect): Bool;

This function determines whether two rectangles are equal by comparing the coordinates of their upper-left and lower-right corners. If the values of these coordinates are equal, **EqualRect** returns a nonzero value; otherwise, it returns zero.

Parameters

lpRect1	**LPRECT** Points to a **RECT** data structure that contains the upper-left and lower-right corner coordinates of the first rectangle.
lpRect2	**LPRECT** Points to a **RECT** data structure that contains the upper-left and lower-right corner coordinates of the second rectangle.

Return value The return value specifies whether the specified rectangles are equal. It is nonzero if the two rectangles are identical. Otherwise, it is zero.

EqualRgn

Syntax BOOL EqualRgn(hSrcRgn1, hSrcRgn2)
function EqualRgn(SrcRgn1, SrcRgn2: HRgn): Bool;

This function checks the two given regions to determine whether they are identical.

Parameters

hSrcRgn1	**HRGN** Identifies a region.
hSrcRgn2	**HRGN** Identifies a region.

Return value The return value specifies whether the specified regions are equal. It is nonzero if the two regions are equal. Otherwise, it is zero.

Escape

Syntax int Escape(hDC, nEscape, nCount, lpInData, lpOutData)
function Escape(DC: HDC; Escape, Count: Integer; InData, OutData: Pointer): Integer;

This function allows applications to access facilities of a particular device that are not directly available through GDI. Escape calls made by an application are translated and sent to the device driver.

Parameters

hDC — **HDC** Identifies the device context.

nEscape — **int** Specifies the escape function to be performed. For a complete list of escape functions, see Chapter 12, "Printer escapes," in *Reference, Volume 2*.

nCount — **int** Specifies the number of bytes of data pointed to by the *lpInData* parameter.

lpInData — **LPSTR** Points to the input data structure required for this escape.

lpOutData — **LPSTR** Points to the data structure to receive output from this escape. The *lpOutData* parameter should be NULL if no data are returned.

Return value The return value specifies the outcome of the function. It is positive if the function is successful except for the QUERYESCSUPPORT escape, which only checks for implementation. The return value is zero if the escape is not implemented. A negative value indicates an error. The following list shows common error values:

Value	Meaning
SP_ERROR	General error.
SP_OUTOFDISK	Not enough disk space is currently available for spooling, and no more space will become available.
SP_OUTOFMEMORY	Not enough memory is available for spooling.
SP_USERABORT	User terminated the job through the Print Manager.

EscapeCommFunction

Syntax int EscapeCommFunction(nCid, nFunc)
function EscapeCommFunction(Cid, Func: Integer): Integer;

This function directs the communication device specified by the *nCid* parameter to carry out the extended function specified by the *nFunc* parameter.

Parameters

nCid — **int** Specifies the communication device to carry out the extended function. The **OpenComm** function returns this value.

nFunc — **int** Specifies the function code of the extended function. It can be any one of the following values:

Value	Description
CLRDTR	Clears the data-terminal-ready (DTR) signal.
CLRRTS	Clears the request-to-send (RTS) signal.
RESETDEV	Resets the device if possible.
SETDTR	Sends the data-terminal-ready (DTR) signal.
SETRTS	Sends the request-to-send (RTS) signal.
SETXOFF	Causes transmission to act as if an XOFF character has been received.
SETXON	Causes transmission to act as if an XON character has been received.

Return value The return value specifies the result of the function. It is zero if it is successful. It is negative if the *nFunc* parameter does not specify a valid function code.

ExcludeClipRect

Syntax int ExcludeClipRect(hDC, X1, Y1, X2, Y2)
function ExcludeClipRect(DC: HDC; X1, Y1, X2, Y2: Integer): Integer;

This function creates a new clipping region that consists of the existing clipping region minus the specified rectangle.

Parameters

hDC — **HDC** Identifies the device context.

X1	**int** Specifies the logical *x*-coordinate of the upper-left corner of the rectangle.
Y1	**int** Specifies the logical *y*-coordinate of the upper-left corner of the rectangle.
X2	**int** Specifies the logical *x*-coordinate of the lower-right corner of the rectangle.
Y2	**int** Specifies the logical *y*-coordinate of the lower-right corner of the rectangle.

Return value The return value specifies the new clipping region's type. It can be any one of the following values:

Value	Meaning
COMPLEXREGION	The region has overlapping borders.
ERROR	No region was created.
NULLREGION	The region is empty.
SIMPLEREGION	The region has no overlapping borders.

Comments The width of the rectangle, specified by the absolute value of *X2* – *X1*, must not exceed 32,767 units. This limit applies to the height of the rectangle as well.

ExcludeUpdateRgn

Syntax int ExcludeUpdateRgn(hDC, hWnd)
function ExcludeUpdateRgn(DC: HDC; Wnd: HWnd): Integer;

This function prevents drawing within invalid areas of a window by excluding an updated region in the window from a clipping region.

Parameters

hDC	**HANDLE** Identifies the device context associated with the clipping region.
hWnd	**HWND** Identifies the window being updated.

Return value This value specifies the type of resultant region. It can be any one of the following values:

Value	Meaning
COMPLEXREGION	The region has overlapping borders.
ERROR	No region was created.
NULLREGION	The region is empty.
SIMPLEREGION	The region has no overlapping borders.

ExitWindows 3.0

Syntax BOOL ExitWindows(dwReserved, wReturnCode)
function ExitWindows(Reserved: Longint; ReturnCode: Word): Bool;

This function initiates the standard Windows shutdown procedure. If all applications agree to terminate, the Windows session is terminated and control returns to DOS. Windows sends the WM_QUERYENDSESSION message to notify all applications that a request has been made to terminate Windows. If all applications agree to terminate, Windows sends the WM_ENDSESSION message to all applications before exiting.

Parameters

dwReserved	**DWORD** Is reserved and should be set to zero.
wReturnCode	**WORD** Specifies the return value to be passed to DOS when Windows exits.

Return value The return value is FALSE if one or more applications refused to terminate. The function does not return if all applications agree to be terminated.

ExtDeviceMode 3.0

Syntax int ExtDeviceMode(hWnd, hDriver, lpDevModeOutput, lpDeviceName, lpPort, lpDevModeInput, lpProfile, wMode)
type TextDeviceMode = function(Wnd: Hwnd; Driver: THandle; var DevModeOutput: TDevMode; DeviceName, Port: PChar; var DevModeInput: TDevMode; Profile: PChar; Mode: Word): Integer;

This function retrieves or modifies device initialization information for a given printer driver, or displays a driver-supplied dialog box for configuring the printer driver. Printer drivers that support device initialization by applications export this **ExtDeviceMode** so that applications can call it.

Parameters

hWnd	**HWND** Identifies a window. If the application calls **ExtDeviceMode** to display a dialog box, the specified window is the parent of the dialog box.
hDriver	**HANDLE** Identifies the device-driver module. The **GetModuleHandle** function or **LoadLibrary** function returns a module handle.
lpDevModeOutput	**DEVMODE FAR** * Points to a **DEVMODE** data structure. The driver writes the initialization information

supplied in the *lpDevModeInput* parameter to this structure.

lpDeviceName — **LPSTR** Points to a null-terminated character string that contains the name of the printer device, such as "PCL/HP LaserJet."

lpPort — **LPSTR** Points to a null-terminated character string that contains the name of the port to which the device is connected, such as LPT1:.

lpDevModeInput — **DEVMODE FAR** * Points to a **DEVMODE** data structure that supplies initialization information to the printer driver.

lpProfile — **LPSTR** Points to a null-terminated string that contains the name of the initialization file which initialization information is recorded in and read from. If this parameter is NULL, WIN.INI is the default.

wMode — **WORD** Specifies a mask of values which determine the types of operations the function will perform. If *wMode* is zero, **ExtDeviceMode** returns the number of bytes required by the printer device driver's **DEVMODE** structure. Otherwise, *wMode* must be one or more of the following values:

Value	**Meaning**
DM_COPY	Writes the printer driver's current print settings to the **DEVMODE** data structure identified by **lpDevMode-Output.** The calling application must allocate a buffer sufficiently large to contain the information. If this bit is clear, *lpDevModeOutput* can be NULL.
DM_MODIFY	Changes the printer driver's current print settings to match the partial initialization data in the **DEVMODE** data structure identified by *lpDevModeInput* before prompting, copying, or updating.
DM_PROMPT	Presents the printer driver's Print Setup dialog box and then changes the current print settings to those the user specifies.

DM_UPDATE	Writes the printer driver's current print settings to the printer environment and the WIN.INI initialization file.

Return value If the *wMode* parameter is zero, the return value is the size of the **DEVMODE** data structure required to contain the printer driver initialization data. If the function displays the initialization dialog box, the return value is either IDOK or IDCANCEL, depending on which button the user selected. If the function does not display the dialog box and was successful, the return value is IDOK. The return value is less than zero if the function failed.

Comments The **ExtDeviceMode** function is actually part of the printer's device driver, and not part of GDI. To call this function, the application must include the DRIVINT.H file, load the printer device driver, and retrieve the address of the function by using the **GetProc-Address** function. The application can then use the address to set up the printer.

An application can set the *wMode* parameter to DM_COPY to obtain a **DEVMODE** data structure filled in with the printer driver's initialization data. The application can then pass this data structure to the **CreateDC** function to set a private environment for the printer device context.

ExtFloodFill 3.0

Syntax BOOL ExtFloodFill(hDC, X, Y, crColor, wFillType)
function ExtFloodFill(DC: HDC; X, Y: Integer; Color: TColorRef; FillType: Word): Bool;

This function fills an area of the display surface with the current brush.

If *wFillType* is set to FLOODFILLBORDER, the area is assumed to be completely bounded by the color specified by the *crColor* parameter. The **ExtFloodFill** function begins at the point specified by the *X* and *Y* parameters and fills in all directions to the color boundary.

If *wFillType* is set to FLOODFILLSURFACE, the **ExtFloodFill** function begins at the point specified by *X* and *Y* and continues in all directions, filling all adjacent areas containing the color specified by *crColor*.

Parameters

hDC	**HDC** Identifies the device context.
X	**int** Specifies the logical *x*-coordinate of the point where filling begins.

Y	**int** Specifies the logical *y*-coordinate of the point where filling begins.
crColor	**COLORREF** Specifies the color of the boundary or of the area to be filled. The interpretation of *crColor* depends on the value of the *wFillType* parameter.
wFillType	**WORD** Specifies the type of flood fill to be performed. It must be one of the following values:

Value	Meaning
FLOODFILLBORDER	The fill area is bounded by the color specified by *crColor*. This style is identical to the filling performed by the **FloodFill** function.
FLOODFILLSURFACE	The fill area is defined by the color specified by *crColor*. Filling continues outward in all directions as long as the color is encountered. This is useful for filling areas with multicolored boundaries.

Return value The return value specifies the outcome of the function. It is nonzero if the function is successful. It is zero if:

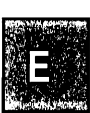
- The filling could not be completed
- The given point has the boundary color specified by *crColor* (if FLOODFILLBORDER was requested)
- The given point does not have the color specified by *crColor* (if FLOODFILLSURFACE was requested)
- The point is outside the clipping region

Comments Only memory device contexts and devices that support raster-display technology support the **ExtFloodFill** function. For more information, see the RC_BITBLT raster capability in the **GetDeviceCaps** function section.

ExtTextOut

Syntax BOOL ExtTextOut(hDC, X, Y, wOptions, lpRect, lpString, nCount, lpDx)
function ExtTextOut(DC: HDC; X, Y: Integer; Options: Word; Rect: PRect; Str: PChar; Count: Word; Dx: PInteger): Bool;

This function writes a character string, within a rectangular region on the specified display, using the currently selected font. The rectangular region can be opaque (filled with the current background color) and it can be a clipping region.



Parameters

hDC	**HDC** Identifies the device context.
X	**int** Specifies the logical *x*-coordinate of the origin of the character cell for the first character in the specified string.
Y	**int** Specifies the logical *y*-coordinate of the origin of the character cell for the first character in the specified string.
wOptions	**WORD** Specifies the rectangle type. It can be one or both of the following values, or neither: ETO_CLIPPED ETO_OPAQUE The ETO_CLIPPED value specifies that Windows will clip text to the rectangle. The ETO_OPAQUE value specifies that the current background color fills the rectangle.
lpRect	**LPRECT** Points to a **RECT** data structure. The *lpRect* parameter can be NULL.
lpString	**LPSTR** Points to the specified character string.
nCount	**int** Specifies the number of characters in the string.
lpDx	**LPINT** Points to an array of values that indicate the distance between origins of adjacent character cells. For instance, *lpDx*[*i*] logical units will separate the origins of character cell *i* and character cell *i* + 1.

Return value The return value specifies whether or not the string is drawn. It is nonzero if the string is drawn. Otherwise, it is zero.

Comments If *lpDx* is NULL, the function uses the default spacing between characters.

The character-cell origins and the contents of the array pointed to by the *lpDx* parameter are given in logical units. A character-cell origin is defined as the upper-left corner of the character cell.

By default, the current position is not used or updated by this function. However, an application can call the **SetTextAlign** function with the *wFlags* parameter set to TA_UPDATECP to permit Windows to use and update the current position each time the application calls **ExtTextOut** for a given device context. When this flag is set, Windows ignores the *X* and *Y* parameters on subsequent **ExtTextOut** calls.

FatalAppExit 3.0

Syntax VOID FatalAppExit(wAction, lpMessageText)
procedure Fatal AppExit(Action: Word; MessageText: PChar);

This function displays a message containing the text specified by the *lpMessageText* parameter and terminates the application when the message box is closed. When called under the debugging version of Windows, the message box gives the user the opportunity to terminate the application or to return to the caller.

Parameters

wAction	**WORD** Is reserved and must be set to 0.
lpMessageText	**LPSTR** Points to a character string that is displayed in the message box. The message is displayed on a single line. To accommodate low-resolution displays, the string should be no more than 35 characters in length.

Return value None.

Comments An application that encounters an unexpected error should terminate by freeing all its memory and then returning from its main message loop. It should call **FatalAppExit** only when it is not capable of terminating any other way. **FatalAppExit** may not always free an application's memory or close its files, and it may cause a general failure of Windows.

FatalExit

Syntax void FatalExit(Code)
procedure FatalExit(Code: Integer);

This function displays the current state of Windows on the debugging monitor and prompts for instructions on how to proceed. The display includes an error code, the *Code* parameter, followed by a symbolic stack trace, showing the flow of execution up to the point of call.

An application should call this function only for debugging purposes; it should not call the function in a retail version of the application. Calling this function in the retail version will terminate the application.

Parameters

Code	**int** Specifies the error code to be displayed.

Return value None.

Comments The **FatalExit** function prompts the user to respond to an "Abort, Break or Ignore" message. **FatalExit** processes the response as follows:

Response	Description
A (Abort)	Terminates Windows.
B (Break)	Simulates a non-maskable interrupt (NMI) to enter the debugger.
I (Ignore)	Returns to the caller.

The **FatalExit** function is for debugging only.

An application should call this function whenever the application detects a fatal error. All input and output is received and transmitted through the computer's auxiliary port (AUX) or through the debugger if a debugger is installed.

FillRect

Syntax

int FillRect(hDC, lpRect, hBrush)
function FillRect(DC: HDC; var Rect: TRect; Brush: HBrush): Integer;

This function fills a given rectangle by using the specified brush. The **FillRect** function fills the complete rectangle, including the left and top borders, but does not fill the right and bottom borders.

Parameters

hDC	**HDC** Identifies the device context.
lpRect	**LPRECT** Points to a **RECT** data structure that contains the logical coordinates of the rectangle to be filled.
hBrush	**HBRUSH** Identifies the brush used to fill the rectangle.

Return value

Although the **FillRect** function return type is an integer, the return value is not used and has no meaning.

Comments

The brush must have been created previously by using either the **CreateHatchBrush**, **CreatePatternBrush**, or **CreateSolidBrush** function, or retrieved using the **GetStockObject** function.

When filling the specified rectangle, the **FillRect** function does not include the rectangle's right and bottom sides. GDI fills a rectangle up to, but does not include, the right column and bottom row, regardless of the current mapping mode.

FillRect compares the values of the **top**, **bottom**, **left**, and **right** fields of the specified rectangle. If **bottom** is less than or equal to **top**, or if **right** is less than or equal to **left**, the rectangle is not drawn.

FillRgn

Syntax BOOL FillRgn(hDC, hRgn, hBrush)
function FillRgn(DC: HDC; Rgn: HRgn; Brush: HBrush): Bool;

This function fills the region specified by the *hRgn* parameter with the brush specified by the *hBrush* parameter.

Parameters

hDC	**HDC** Identifies the device context.
hRgn	**HRGN** Identifies the region to be filled. The coordinates for the given region are specified in device units.
hBrush	**HBRUSH** Identifies the brush to be used to fill the region.

Return value The return value specifies the outcome of the function. It is nonzero if the function is successful. Otherwise, it is zero.

FindAtom

Syntax ATOM FindAtom(lpString)
function FindAtom(Str: PChar): TAtom;

This function searches the atom table for the character string pointed to by the *lpString* parameter and retrieves the atom associated with that string.

Parameters

lpString	**LPSTR** Points to the character string to be searched for. The string must be null-terminated.

Return value The return value identifies the atom associated with the given string. It is NULL if the string is not in the table.

FindResource

Syntax HANDLE FindResource(hInstance, lpName, lpType)
function FindResource(Instance: THandle; Name, ResType: PChar): THandle;

This function determines the location of a resource in the specified resource file. The *lpName* and *lpType* parameters define the resource name and type, respectively.

Parameters

hInstance	**HANDLE** Identifies the instance of the module whose executable file contains the resource.

lpName **LPSTR** Points to a null-terminated character string that represents the name of the resource.

lpType **LPSTR** Points to a null-terminated character string that represents the type name of the resource. For predefined resource types, the *lpType* parameter should be one of the following values:

Value	Meaning
RT_ACCELERATOR	Accelerator table
RT_BITMAP	Bitmap resource
RT_DIALOG	Dialog box
RT_FONT	Font resource
RT_FONTDIR	Font directory resource
RT_MENU	Menu resource
RT_RCDATA	User-defined resource (raw data)

Return value The return value identifies the named resource. It is NULL if the requested resource cannot be found.

Comments An application must not call **FindResource** and the **LoadResource** function to load cursor, icon, and string resources. Instead, it must load these resources by calling the following functions:

- **LoadCursor**
- **LoadIcon**
- **LoadString**

An application can call **FindResource** and **LoadResource** to load other predefined resource types. However, it is recommended that the application load the corresponding resources by calling the following functions:

- **LoadAccelerators**
- **LoadBitmap**
- **LoadMenu**

If the high-order word of the *lpName* or *lpType* parameter is zero, the low-order word specifies the integer ID of the name or type of the given resource. Otherwise, the parameters are long pointers to null-terminated character strings. If the first character of the string is a pound sign (#), the remaining characters represent a decimal number that specifies the integer ID of the resource's name or type. For example, the string #258 represents the integer ID 258.

To reduce the amount of memory required for the resources used by an application, the application should refer to the resources by integer ID instead of by name.

FindWindow

Syntax HWND FindWindow(lpClassName, lpWindowName)
function FindWindow(ClassName, WindowName: PChar): HWnd;

This function returns the handle of the window whose class is given by the *lpClassName* parameter and whose window name, or caption, is given by the *lpWindowName* parameter. This function does not search child windows.

Parameters

lpClassName — **LPSTR** Points to a null-terminated character string that specifies the window's class name. If *lpClassName* is NULL, all class names match.

lpWindowName — **LPSTR** Points to a null-terminated character string that specifies the window name (the window's text caption). If *lpWindowName* is NULL, all window names match.

Return value The return value identifies the window that has the specified class name and window name. It is NULL if no such window is found.

FlashWindow

Syntax BOOL FlashWindow(hWnd, bInvert)
function FlashWindow(Wnd: HWnd; Invert: Bool): Bool;

This function "flashes" the given window once. Flashing a window means changing the appearance of its caption bar as if the window were changing from inactive to active status, or vice versa. (An inactive caption bar changes to an active caption bar; an active caption bar changes to an inactive caption bar.)

Typically, a window is flashed to inform the user that the window requires attention, but that it does not currently have the input focus.

Parameters

hWnd — **HWND** Identifies the window to be flashed. The window can be either open or iconic.

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

Return value The return value specifies the window's state before call to the **FlashWindow** function. It is nonzero if the window was active before the call. Otherwise, it is zero.

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

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

This function always returns a nonzero value for iconic windows. If the window is iconic, **FlashWindow** will simply flash the icon; *bInvert* is ignored for iconic windows.

FloodFill

Syntax BOOL FloodFill(hDC, X, Y, crColor)
function FloodFill(DC: HDC; X, Y: Integer; Color: TColorRef): Bool;

This function fills an area of the display surface with the current brush. The area is assumed to be bounded as specified by the *crColor* parameter. The **FloodFill** function begins at the point specified by the *X* and *Y* parameters and continues in all directions to the color boundary.

Parameters

hDC	**HDC** Identifies the device context.
X	**int** Specifies the logical *x*-coordinate of the point where filling begins.
Y	**int** Specifies the logical *y*-coordinate of the point where filling begins.
crColor	**COLORREF** Specifies the color of the boundary.

Return value The return value specifies the outcome of the function. It is nonzero if the function is successful. It is zero if the filling could not be completed, the given point has the boundary color specified by *crColor*, or the point is outside the clipping region.

Comments Only memory device contexts and devices that support raster-display technology support the **FloodFill** function. For more information, see the RC_BITBLT raster capability in the **GetDeviceCaps** function, later in this chapter.

FlushComm

Syntax intFlushComm(nCid, nQueue)
function FlushComm(Cid, Queue: Integer): Integer;

This function flushes all characters from the transmit or receive queue of the communication device specified by the *nCid* parameter. The *nQueue* parameter specifies which queue is to be flushed.

Parameters

nCid	**int** Specifies the communication device to be flushed. The **OpenComm** function returns this value.
nQueue	**int** Specifies the queue to be flushed. If *nQueue* is zero, the transmit queue is flushed. If it is 1, the receive queue is flushed.

Return value The return value specifies the result of the function. It is zero if it is successful. It is negative if *nCid* is not a valid device, or if *nQueue* is not a valid queue.

_FPInit

Syntax void far * _FPInit()

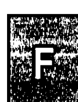
This function initializes the Windows floating-point emulator library (WIN87EM.DLL) or floating-point coprocessor and sets up a default floating-point exception-handler routine. Only DLLs need to call this function.

Parameters None.

Return value The return value is a pointer to the previous floating-point exception handler.

Comments A DLL must ensure that the floating-point emulator or coprocessor has beeninitialized before making any function calls that use floating-pointarithmetic. If a task that does not initialize the floating-pointemulator or coprocesoor can call the DLL, or if the task's floating-point exception handler does not handle floating-point exceptions appropriately for the DLL, the DLL must call the **_FPInit** function to initialize the emulator or coprocessor. Before returning control to the calling task, the DLL must call the **_FPTerm** function to restore the previous exception handler.

_FPTerm



Syntax void _FPTerm(lpOldFPSigHandler)

This function restores the floating-point exception-handler routine that was in effect when a DLL called the **_FPInit** function to initialize the floating-point emulator or coprocessor. Only DLLs need to call this function.

Parameters

lpOldFPSigHandler	**void far *** Points to the floating-point exception handler to be restored.

Return value None.

Comments A DLL must ensure that the floating-point emulator or coprocessor has been initialized before making any function calls that use floating-point arithmetic. If a task that does not initialize the floating-point emulator or coprocessor can call the DLL, or if it is possible that the task's floating-point exception handler does not handle floating-point exceptions appropriately for the DLL, the DLL must call the **_FPInit** function to initialize the emulator or coprocessor. Before returning control to the calling task, the DLL must call the **_FPTerm** function to restore the previous exception handler.

FrameRect

Syntax int FrameRect(hDC, lpRect, hBrush)
procedure FrameRect(DC: HDC; var Rect: TRect; Brush: HBrush;

This function draws a border around the rectangle specified by the *lpRect* parameter. The **FrameRect** function uses the given brush to draw the border. The width and height of the border is always one logical unit.

Parameters

hDC	**HDC** Identifies the device context of the window.
lpRect	**LPRECT** Points to a **RECT** data structure that contains the logical coordinates of the upper-left and lower-right corners of the rectangle.
hBrush	**HBRUSH** Identifies the brush to be used for framing the rectangle.

Return value Although the return value type is integer, its contents should be ignored.

Comments The brush identified by the *hBrush* parameter must have been created previously by using the **CreateHatchBrush**, **CreatePatternBrush**, or **CreateSolidBrush** function.

If the **bottom** field is less than or equal to the **top** field, or if **right** is less than or equal to **left**, the rectangle is not drawn.

FrameRgn

Syntax BOOL FrameRgn(hDC, hRgn, hBrush, nWidth, nHeight)
function FrameRgn(DC: HDC; Rgn: HRgn; Brush: HBrush; Width, Height: Integer): Bool;

This function draws a border around the region specified by the *hRgn* parameter, using the brush specified by the *hBrush* parameter. The *nWidth* parameter specifies the width of the border in vertical brush strokes; the *nHeight* parameter specifies the height in horizontal brush strokes.

Parameters

hDC	**HDC** Identifies the device context.
hRgn	**HANDLE** Identifies the region to be enclosed in a border. The coordinates for the given region are specified in device units.
hBrush	**HBRUSH** Identifies the brush to be used to draw the border.
nWidth	**int** Specifies the width in vertical brush strokes (in logical units).
nHeight	**int** Specifies the height in horizontal brush strokes (in logical units).

Return value The return value specifies the outcome of the function. It is nonzero if the function is successful. Otherwise, it is zero.

FreeLibrary

Syntax void FreeLibrary(hLibModule)
procedure FreeLibrary(LibModule: THandle);

This function decreases the reference count of the loaded library module by one. When the reference count reaches zero, the memory occupied by the module is freed.

Parameters *hLibModule* **HANDLE** Identifies the loaded library module.

Return value None.

Comments A DLL must not call the **FreeLibrary** function within its WEP function.

FreeModule 3.0

Syntax
void FreeModule(hModule)
function FreeModule(Module: THandle): Bool;

This function decreases the reference count of the loaded module by one. When the reference count reaches zero, the memory occupied by the module is freed.

Parameters *hModule* **HANDLE** Identifies the loaded module.

Return value None.

FreeProcInstance

Syntax
void FreeProcInstance(lpProc)
procedure FreeProcInstance(Proc: TFarProc);

This function frees the function specified by the *lpProc* parameter from the data segment bound to it by the **MakeProcInstance** function.

Parameters *lpProc* **FARPROC** Is the procedure-instance address of the function to be freed. It must have been created previously by using the **MakeProcInstance** function.

Return value None.

Comments After freeing a procedure instance, attempts to call the function using the freed procedure-instance address will result in an unrecoverable error.

FreeResource

Syntax
BOOL FreeResource(hResData)
function FreeResource(ResData: THandle): Bool;

This function removes a loaded resource from memory by freeing the allocated memory occupied by that resource.

The **FreeResource** function does not actually free the resource until the reference count is zero (that is, the number of calls to the function equals the number of times the application called the **LoadResource** function for this resource). This ensures that the data remain in memory for the application to use.

Parameters *hResData* **HANDLE** Identifies the data associated with the resource. The handle is assumed to have been created by using the **LoadResource** function.

Return value The return value specifies the outcome of the function. The return value is nonzero if the function has failed and the resource has not been freed. The return value is zero if the function is successful.

FreeSelector 3.0

Syntax WORD FreeSelector(wSelector)
function FreeSelector(Selector: Word): Word;

This function frees a selector originally allocated by the AllocSelector or AllocDStoCSAlias functions. After the application calls this function, the selector is invalid and must not be used.

Parameters *wSelector* **WORD** Specifies the selector to be freed.

Return value The return value is NULL if the function was successful. Otherwise, it is the selector specified by the *wSelector* parameter.

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

GetActiveWindow

Syntax HWND GetActiveWindow( )
function GetActiveWindow: HWnd;

This function retrieves the window handle of the active window. The active window is either the window that has the current input focus, or the window explicitly made active by the **SetActiveWindow** function.

Parameters None.

Return value The return value identifies the active window.

GetAspectRatioFilter

Syntax DWORD GetAspectRatioFilter(hDC)
function GetAspectRatioFilter(DC: HDC): Longint;

This function retrieves the setting for the current aspect-ratio filter. The aspect ratio is the ratio formed by a device's pixel width and height. Information about a device's aspect ratio is used in the creation, selection, and displaying of fonts. Windows provides a special filter, the aspect-ratio filter, to select fonts designed for a particular aspect ratio from all of the available fonts. The filter uses the aspect ratio specified by the **SetMapperFlags** function.

Parameters *hDC* **HDC** Identifies the device context that contains the specfied aspect ratio.

Return value The return value specifies the aspect ratio used by the current aspect-ratio filter. The *x*-coordinate of the aspect ratio is contained in the high-order word, and the *y*-coordinate is contained in the low-order word.

GetAsyncKeyState

Syntax int GetAsyncKeyState(vKey)
function GetAsyncKeyState(Key: Integer): Integer;

This function determines whether a key is up or down at the time the function is called, and whether the key was pressed after a previous call to the **GetAsyncKeyState** function. If the most significant bit of the return value is set, the key is currently down; if the least significant bit is set, the key was pressed after a previous call to the function.

Parameters *vkey* **int** Specifies one of 256 posible virtual-key code values.

Return value The return value specifies whether the key was pressed since the last call to **GetAsyncKeyState** and whether the key is currently up or down. If the most significant bit is set, the key is down, and if the least significant bit is set, the key was pressed after a preceding **GetAsyncKeyState** call.

GetAtomHandle

Syntax HMEM GetAtomHandle(wAtom)
function GetAtomHandle(AnAtom: TAtom): THandle;

This function retrieves a handle (relative to the local heap) of the string that corresponds to the atom specified by the *wAtom* parameter.

Parameters *wAtom* **WORD** Specifies an unsigned integer that identifies the atom whose handle is to be retrieved.

Return value The return value identifies the given atom's string. It is zero if no such atom exists.

GetAtomName

Syntax WORD GetAtomName(nAtom, lpBuffer, nSize)
function GetAtomName(AnAtom: TAtom; Buffer: PChar; Size: Integer): Word;

This function retrieves a copy of the character string associated with the *nAtom* parameter and places it in the buffer pointed to by the *lpBuffer* parameter. The *nSize* parameter specifies the maximum size of the buffer.

Parameters

nAtom **ATOM** Identifies the character string to be retrieved.

lpBuffer **LPSTR** Points to the buffer that is to receive the character string.

nSize **int** Specifies the maximum size (in bytes) of the buffer.

Return value The return value specifies the actual number of bytes copied to the buffer. It is zero if the specified atom is not valid.

GetBitmapBits

Syntax DWORD GetBitmapBits(hBitmap, dwCount, lpBits)
function GetBitmapBits(Bitmap: HBitmap; Count: Longint; Bits: Pointer): Longint;

This function copies the bits of the specified bitmap into the buffer that is pointed to by the *lpBits* parameter. The *dwCount* parameter specifies the number of bytes to be copied to the buffer. The **GetObject** function should be used to determine the correct *dwCount* value for the given bitmap.

Parameters

hBitmap **HBITMAP** Identifies the bitmap.

dwCount **DWORD** Specifies the number of bytes to be copied.

lpBits **LPSTR** Long pointer to the buffer that is to receive the bitmap. The bitmap is an array of bytes. The bitmap byte

array conforms to a structure where horizontal scan lines are multiples of 16 bits.

Return value The return value specifies the actual number of bytes in the bitmap. It is zero if there is an error.

GetBitmapDimension

Syntax DWORD GetBitmapDimension(hBitmap)
function GetBitmapDimension(Bitmap: HBitmap): Longint;

This function returns the width and height of the bitmap specified by the *hBitmap* parameter. The height and width is assumed to have been set previously by using the **SetBitmapDimension** function.

Parameters *hBitmap* **HBITMAP** Identifies the bitmap.

Return value The return value specifies the width and height of the bitmap, measured in tenths of millimeters. The height is in the high-order word, and the width is in the low-order word. If the bitmap width and height have not been set by using **SetBitmapDimension**, the return value is zero.

GetBkColor

Syntax DWORD GetBkColor(hDC)
function GetBkColor(DC: HDC): Longint;

This function returns the current background color of the specified device.

Parameters *hDC* **HDC** Identifies the device context.

Return value The return value specifies an RGB color value that names the current background color.

GetBkMode

Syntax int GetBkMode(hDC)
function GetBkMode(DC: HDC): Integer;

This function returns the background mode of the specified device. The background mode is used with text, hatched brushes, and pen style that is not a solid line.

Parameters *hDC* **HDC** Identifies the device context.

Return value The return value specifies the current background mode. It can be OPAQUE or TRANSPARENT.

GetBrushOrg

Syntax DWORD GetBrushOrg(hDC)
function GetBrushOrg(DC: HDC): Longint;

This function retrieves the current brush origin for the given device context.

Parameters *hDC* **HDC** Identifies the device context.

Return value The return value specifies the current origin of the brush. The *x*-coordinate is in the low word, and the *y*-coordinate is in the high word. The coordinates are assumed to be in device units.

Comments The initial brush origin is at the coordinate (0,0).

GetBValue

Syntax BYTE GetBValue(rgbColor)
function GetBValue(RGBColor: Longint): Byte;

This macro extracts the blue value from an RGB color value.

Parameters *rgbColor* **DWORD** Specifies a red, a green, and a blue color field, each specifying the intensity of the given color.

Return value The return value specifies a byte that contains the blue value of the *rgbColor* parameter.

Comments The value 0FFH corresponds to the maximum intensity value for a single byte; 000H corresponds to the minimum intensity value for a single byte.

GetCapture

Syntax HWND GetCapture( )
function GetCapture: HWnd;

This function retrieves a handle that identifies the window that has the mouse capture. Only one window has the mouse capture at any given time; this window receives mouse input whether or not the cursor is within its borders.

Parameters None.

Return value The return value identifies the window that has the mouse capture; it is NULL if no window has the mouse capture.

Comments A window receives the mouse capture when its handle is passed as the *hWnd* parameter of the **SetCapture** function.

GetCaretBlinkTime

Syntax WORD GetCaretBlinkTime( )
function GetCaretBlinkTime: Word;

This function retrieves the caret blink rate. The blink rate is the elapsed time in milliseconds between flashes of the caret.

Parameters None.

Return value The return value specifies the blink rate (in milliseconds).

GetCaretPos

Syntax void GetCaretPos(lpPoint)
procedure GetCaretPos(var Point: TPoint);

This function retrieves the caret's current position (in screen coordinates), and copies them to the **POINT** structure pointed to by the *lpPoint* parameter.

Parameters *lpPoint* **LPPOINT** Points to the **POINT** structure that is to receive the screen coordinates of the caret.

Return value None.

Comments The caret position is always given in the client coordinates of the window that contains the caret.

GetCharWidth

Syntax BOOL GetCharWidth(hDC, wFirstChar, wLastChar, lpBuffer)
function GetCharWidth(DC: HDC; FirstChar, LastChar: Word; var Buffer): Bool;

This function retrieves the widths of individual characters in a consecutive group of characters from the current font. For example, if the *wFirstChar* parameter identifies the letter *a* and the *wLastChar* parameter identifies the letter *z*, the **GetCharWidth** function retrieves the widths of all lowercase characters. The function stores the values in the buffer pointed to by the *lpBuffer* parameter.

Parameters

hDC	**HDC** Identifies the device context.
wFirstChar	**WORD** Specifies the first character in a consecutive group of characters in the current font.
wLastChar	**WORD** Specifies the last character in a consecutive group of characters in the current font.
lpBuffer	**LPINT** Points to a buffer that will receive the width values for a consecutive group of characters in the current font.

Return value The return value specifies the outcome of the function. It is nonzero if the function is successful. Otherwise, it is zero.

Comments If a character in the consecutive group of characters does not exist in a particular font, it will be assigned the width value of the default character.

GetClassInfo 3.0

Syntax

```
BOOL GetClassInfo(hInstance, lpClassName, lpWndClass)
function GetClassInfo(Instance: THandle; ClassInfo: PChar; var
WndClass: TWndClass): Bool;
```

This function retrieves information about a window class. The *hInstance* parameter identifies the instance of the application that created the class, and the *lpClassName* parameter identifies the window class. If the function locates the specified window class, it copies the **WNDCLASS** data used to register the window class to the **WNDCLASS** data structure pointed to by *lpWndClass*.

Parameters

hInstance	**HANDLE** Identifies the instance of the application that created the class. To retrieve information on classes defined by Windows (such as buttons or list boxes), set *hInstance* to NULL.
lpClassName	**LPSTR** Points to a null-terminated string that contains the name of the class to find. If the high-order word of this parameter is NULL, the low-order word is assumed to be a

value returned by the **MAKEINTRESOURCE** macro used when the class was created.

lpWndClass **LPWNDCLASS** Points to the **WNDCLASS** structure to which the function will copy the class information.

Return value The return value is TRUE if the function found a matching class and successfully copied the data; the return value is FALSE if the function did not find a matching class.

Comments The **lpszClassName, lpszMenuName,** and **hInstance** fields in the **WNDCLASS** data structure are *not* returned by this function. The menu name is not stored internally and cannot be returned. The class name is already known since it is passed to this function. The **GetClassInfo** function returns all other fields with the values used when the class was registered.

G

GetClassLong

Syntax LONG GetClassLong(hWnd, nIndex)
function GetClassLong(Wnd: HWnd; Index: Integer): Longint;

This function retrieves the long value specified by the *nIndex* parameter from the **WNDCLASS** structure of the window specified by the *hWnd* parameter.

Parameters

hWnd **HWND** Identifies the window.

nIndex **int** Specifies the byte offset of the value to be retrieved. It can also be the following value:

Value	Meaning
GCL_WNDPROC	Retrieves a long pointer to the window function.

Return value The return value specifies the value retrieved from the **WNDCLASS** structure.

Comments To access any extra four-byte values allocated when the window-class structure was created, use a positive byte offset as the index specified by the *nIndex* parameter. The first four-byte value in the extra space is at offset zero, the next four-byte value is at offset 4, and so on.

GetClassName

Syntax int GetClassName(hWnd, lpClassName, nMaxCount)
function GetClassName(Wnd: HWnd; ClassName: PChar; MaxCount: Integer): Integer;

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

Parameters

hWnd — **HWND** Identifies the window whose class name is to be retrieved.

lpClassName — **LPSTR** Points to the buffer that is to receive the class name.

nMaxCount — **int** Specifies the maximum number of bytes to be stored in the *lpClassName* parameter. If the actual name is longer, a truncated name is copied to the buffer.

Return value The return value specifies the number of characters actually copied to *lpClassName*. The return value is zero if the specified class name is not valid.

GetClassWord

Syntax WORD GetClassWord(hWnd, nIndex)
function GetClassWord(Wnd: HWnd, Index: Integer): Word;

This function retrieves the word that is specified by the *nIndex* parameter from the **WNDCLASS** structure of the window specified by the *hWnd* parameter.

Parameters

hWnd — **HWND** Identifies the window.

nIndex — **int** Specifies the byte offset of the value to be retrieved. It can also be one of the following values:

Value	Meaning
GCW_CBCLSEXTRA	Tells how many bytes of additional class information you have. For information on how to access this memory, see the following "Comments" section.
GCW_CBWNDEXTRA	Tells how many bytes of additional window information you have. For

	information on how to access this memory, see the following "Comments" section.
GCW_HBRBACKGROUND	Retrieves a handle to the background brush.
GCW_HCURSOR	Retrieves a handle to the cursor.
GCW_HICON	Retrieves a handle to the icon.
GCW_HMODULE	Retrieves a handle to the module.
GCW_STYLE	Retrieves the window-class style bits.

Return value The return value specifies the value retrieved from the **WNDCLASS** structure.

Comments To access any extra two-byte values allocated when the window-class structure was created, use a positive byte offset as the index specified by the *nIndex* parameter, starting at zero for the first two-byte value in the extra space, 2 for the next two-byte value and so on.

GetClientRect

Syntax void GetClientRect(hWnd, lpRect)
procedure GetClientRect(Wnd: HWnd; var Rect: TRect);

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

Parameters

hWnd	**HWND** Identifies the window associated with the client area.
lpRect	**LPRECT** Points to a **RECT** data structure.

Return value None.

GetClipboardData

Syntax HANDLE GetClipboardData(wFormat)
function GetClipboardData(Format: Word): THandle;

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

Parameters *wFormat* **WORD** Specifies a data format. For a description of the data formats, see the **SetClipboardData** function, later in this chapter.

Return value The return value identifies the memory block that contains the data from the clipboard. The handle type depends on the type of data specified by the *wFormat* parameter. It is NULL if there is an error.

Comments The available formats can be enumerated in advance by using the **EnumClipboardFormats** function.

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

Windows supports two formats for text, CF_TEXT and CF_OEMTEXT. CF_TEXT is the default Windows text clipboard format, while Windows uses the CF_OEMTEXT format for text in non-Windows applications. If you call **GetClipboardData** to retrieve data in one text format and the other text format is the only available text format, Windows automatically converts the text to the requested format before supplying it to your application.

If the clipboard contains data in the CF_PALETTE (logical color palette) format, the application should assume that any other data in the clipboard is realized against that logical palette.

GetClipboardFormatName

Syntax

```
int GetClipboardFormatName(wFormat, lpFormatName, nMaxCount)
function GetClipboardFormatName(Format: Word; FormatName: PChar;
MaxCount: Integer): Integer;
```

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

Parameters

wFormat **WORD** Specifies the type of format to be retrieved. It must not specify any of the predefined clipboard formats.

lpFormatName **LPSTR** Points to the buffer that is to receive the format name.

nMaxCount	**int** Specifies the maximum length (in bytes) of the string to be copied to the buffer. If the actual name is longer, it is truncated.

Return value The return value specifies the actual length of the string copied to the buffer. It is zero if the requested format does not exist or is a predefined format.

GetClipboardOwner

Syntax HWND GetClipboardOwner( )
function GetClipboardOwner: HWnd;

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

Parameters None.

Return value The return value identifies the window that owns the clipboard. It is NULL if the clipboard is not owned.

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

GetClipboardViewer

Syntax HWND GetClipboardViewer( )
function GetClipboardViewer: HWnd;

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

Parameters None.

Return value The return value identifies the window currently responsible for displaying the clipboard. It is NULL if there is no viewer.

GetClipBox

Syntax int GetClipBox(hDC, lpRect)
function GetClipBox(DC: HDC; var Rect: TRect): Integer;

This function retrieves the dimensions of the tightest bounding rectangle around the current clipping boundary. The dimensions are copied to the buffer pointed to by the *lpRect* parameter.

Parameters

hDC — **HDC** Identifies the device context.

lpRect — **LPRECT** Points to the **RECT** data structure that is to receive the rectangle dimensions.

Return value The return value specifies the clipping region's type. It can be any one of the following values:

Value	Meaning
COMPLEXREGION	Clipping region has overlapping borders.
ERROR	Device context is not valid.
NULLREGION	Clipping region is empty.
SIMPLEREGION	Clipping region has no overlapping borders.

GetCodeHandle

Syntax

HANDLE GetCodeHandle(lpProc)
function GetCodeHandle(Proc: TFarProc): THandle;

This function determines which code segment contains the function pointed to by the *lpProc* parameter.

Parameters

lpProc — **FARPROC** Is a procedure-instance address.

Return value The return value identifies the code segment that contains the function.

Comments If the code segment that contains the function is already loaded, the **GetCodeHandle** function marks the segment as recently used. If the code segment is not loaded, **GetCodeHandle** attempts to load it. Thus, an application can use this function to attempt to preload one or more segments needed to perform a particular task.

GetCodeInfo 3.0

Syntax

void GetCodeInfo(lpProc, lpSegInfo)
procedure GetCodeInfo(Proc: TFarProc; SegInfo: Pointer);

This function retrieves a pointer to an array of 16-bit values containing information about the code segment that contains the function pointed to by the *lpProc* parameter.

Parameters

lpProc — **FARPROC** Is the address of the function in the segment for which information is to be retrieved. Instead of a segment:offset address, this value can also be in the form of a module handle and segment number. The **GetModuleHandle** function returns the handle of a named module.

lpSegInfo — **LPVOID** Points to an array of four 32-bit values that will be filled with information about the code segment. See the following "Comments" section for a description of the values in this array.

Return value None.

Comments The *lpSegInfo* parameter points to an array of four 32-bit values that contains such information as the location and size of the segment and its attributes. The following list describes each of these values:

Offset	Description
0	Specifies the logical-sector offset (in bytes) to the contents of the segment data, relative to the beginning of the file. Zero means no file data is available.
2	Specifies the length of the segment in the file (in bytes). Zero means 64K.
4	Contains flags which specify attributes of the segment. The following list describes these flags:

Bit	Meaning
0–2	Specifies the segment type. If bit 0 is set to 1, the segment is a data segment. Otherwise, the segment is a code segment.
3	Specifies whether segment data is iterated. When this bit set to 1, the segment data is iterated.
4	Specifies whether the segment is moveable or fixed. When this bit is set to 1, the segment is moveable. Otherwise, it is fixed.
5	Is not returned.
6	Is not returned.
7	Specifies whether the segment is a read-only data segment or an execute-only code segment. If this bit is set to 1 and the segment is a code segment, the segment is an execute-only segment. If this bit is set to zero and the segment is a data segment, it is a read-only segment.
8	Specifies whether the segment has associated relocation information. If this bit is set to 1, the segment has relocation information. Otherwise, the segment does not have relocation information.
9	Specifies whether the segment has debugging information. If this bit is set to 1, the segment has debugging

		information. Otherwise, the segment does not have debugging information.
	10–11	Is not returned.
	12–15	Is not returned.
6	Specifies the total amount of memory allocated for the segment. This amount may exceed the actual size of the segment. Zero means 65,536.	

GetCommError

Syntax int GetCommError(nCid, lpStat)
function GetCommError(Cid: Integer; var Stat: TComStat): Integer;

In case of a communications error, Windows locks the communications port until the error is cleared by using the **GetCommError** function. This function fills the status buffer pointed to by the *lpStat* parameter with the current status of the communication device specified by the *nCid* parameter. It also returns the error codes that have occurred since the last **GetCommError** call. If *lpStat* is NULL, only the error code is returned. For a list of the error codes, see Table 4.8, "Communications error codes."

Parameters

nCid — **int** Specifies the communication device to be examined. The **OpenComm** function returns this value.

lpStat — **COMSTAT FAR** * Points to the **COMSTAT** structure that is to receive the device status. The structure contains information about a communication device.

Return value The return value specifies the error codes returned by the most recent communications function. It can be a combination of one or more of the values given in Table 4.8.

Table 4.8 Communications error codes

Value	Meaning
CE_BREAK	The hardware detects a break condition.
CE_CTSTO	Clear-to-send timeout. CTS is low for the duration specified by **CtsTimeout** while trying to transmit a character.
CE_DNS	The parallel device is not selected.
CE_DSRTO	Data-set-ready timeout. DSR is low for the duration specified by **DsrTimeout** while trying to transmit a character.
CE_FRAME	The hardware detects a framing error.
CE_IOE	An I/O error occurs while trying to communicate with a parallel device.
CE_MODE	Requested mode is not supported, or the *nCid* parameter is invalid. If set, this is the only valid error.
CE_OOP	The parallel device signals that it is out of paper.
CE_OVERRUN	A character is not read from the hardware before the next character arrives. The character is lost.
CE_PTO	Timeout occurs when communicating with a parallel device.

Table 4.8: Communications error codes (continued)

CE_RLSDTO	Receive-line-signal-detect timeout. RLSD is low for the duration specified by **RlsdTimeout** while trying to transmit a character.
CE_RXOVER	Receive queue overflow. There is either no room in the input queue or a character is received after the **EofChar** character.
CE_RXPARITY	The hardware detects a parity error.
CE_TXFULL	The transmit queue is full while trying to queue a character.

GetCommEventMask

Syntax WORD GetCommEventMask(nCid, nEvtMask)
function GetCommEventMask(Cid, EvtMask: Integer): Word;

This function retrieves the value of the current event mask, and then clears the mask. This function must be used to prevent loss of an event.

Parameters

nCid — **int** Specifies the communication device to be examined. The **OpenComm** function returns this value.

nEvtMask — **int** Specifies which events are to be enabled. For a list of the event values, see the **SetCommEventMask** function, later in this chapter.

Return value The return value specifies the current event-mask value. Each bit in the event mask specifies whether a given event has occurred. A bit is set to 1 if the event has occurred.

GetCommState

Syntax int GetCommState(nCid, lpDCB)
function GetCommState(Cid: Integer; var DCB: TDCB): Integer;

This function fills the buffer pointed to by the *lpDCB* parameter with the device control block of the communication device specified by the *nCid* parameter.

Parameters

nCid — **int** Specifies the device to be examined. The **OpenComm** function returns this value.

lpDCB — **DCB FAR** *Points to the **DCB** data structure that is to receive the current device control block. The structure defines the control setting for the device.

Return value The return value specifies the outcome of the function. It is zero if the function was successful. If an error occurred, the return value is negative.

GetCurrentPDB 3.0

Syntax WORD GetCurrentPDB()
function GetCurrentPDB: Word;

This function returns the paragraph address or selector of the current DOS Program Data Base (PDB), also known as the Program Segment Prefix (PSP).

Parameters None.

Return value The return value is the paragraph address or selector of the current PDB.

GetCurrentPosition

Syntax DWORD GetCurrentPosition(hDC)
function GetCurrentPosition(DC: HDC): Longint;

This function retrieves the logical coordinates of the current position.

Parameters *hDC* **HDC** Identifies a device context.

Return value The return value specifies the current position. The *y*-coordinate is in the high-order word; the *x*-coordinate is in the low-order word.

GetCurrentTask

Syntax HANDLE GetCurrentTask( )
function GetCurrentTask :THandle;

This function returns the handle of the currently executing task.

Parameters None.

Return value The return value identifies the task if the function is successful. Otherwise, it is NULL.

GetCurrentTime

Syntax DWORD GetCurrentTime( )
function GetCurrentTime: Longint;

This function retrieves the current Windows time. Windows time is the number of milliseconds that have elapsed since the system was booted.

Parameters None.

Return value The return value specifies the current time (in milliseconds).

Comments The **GetCurrentTime** and **GetMessageTime** functions return different times. **GetMessageTime** returns the Windows time when the given message was created, not the current Windows time.

The system timer eventually overflows and resets to zero.

GetCursorPos

Syntax void GetCursorPos(lpPoint)
procedure GetCursorPos(var Point: TPoint);

This function retrieves the cursor's current position (in screen coordinates), that copies them to the **POINT** structure pointed to by the *lpPoint* parameter.

Parameters *lpPoint* **LPPOINT** Points to the **POINT** structure that is to receive the screen coordinates of the cursor.

Return value None

Comments The cursor position is always given in screen coordinates and is not affected by the mapping mode of the window that contains the cursor.

GetDC

Syntax HDC GetDC(hWnd)
function GetDC(Wnd: HWnd): HDC;

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

The **GetDC** function retrieves a common, class, or private display context depending on the class style specified for the given window. For common display contexts, **GetDC** assigns default attributes to the context each time it is retrieved. For class and private contexts, **GetDC** leaves the previously assigned attributes unchanged.

Parameters *hWnd* **HWND** Identifies the window whose display context is to be retrieved.

Return value The return value identifies the display context for the given window's client area if the function is successful. Otherwise, it is NULL.

Comments After painting with a common display context, the **ReleaseDC** function must be called to release the context. Class and private display contexts do not have to be released. Since only five common display contexts are available at any given time, failure to release a display context can prevent other applications from accessing a display context.

GetDCOrg

Syntax

```
DWORD GetDCOrg(hDC)
function GetDCOrg(DC: HDC): Longint;
```

This function obtains the final translation origin for the device context. The final translation origin specifies the offset used by Windows to translate device coordinates into client coordinates for points in an application's window. The final translation origin is relative to the physical origin of the screen display.

Parameters *hDC* **HDC** Identifies the device context whose origin is to be retrieved.

Return value The return value specifies the final translation origin (in device coordinates). The *y*-coordinate is in the high-order word; the *x*-coordinate is in the low-order word.

GetDesktopWindow 3.0

Syntax

```
HWND GetDesktopWindow( )
function GetDesktopWindow: HWnd;
```

This function returns the window handle to the Windows desktop window. The desktop window covers the entire screen and is the area on top of which all icons and other windows are painted.

Parameters None.

Return value The return value identifies the Windows desktop window.

GetDeviceCaps

Syntax int GetDeviceCaps(hDC, nIndex)
function GetDeviceCaps(DC: HDC; Index: Integer): Integer;

This function retrieves device-specific information about a given display device. The *nIndex* parameter specifies the type of information desired.

Parameters

hDC	**HDC** Identifies the device context.
nIndex	**int** Specifies the item to return. It can be any one of the values given in Table 4.9, "GDI information indexes."

Return value The return value specifies the value of the desired item.

Comments Table 4.9 lists the values for the *nIndex* parameter:

Table 4.9 GDI information indexes

Index	Meaning
DRIVERVERSION	Version number; for example, 0x100 for 1.0.
TECHNOLOGY	Device technology. It can be any one of these values: **Value** — **Meaning** DT_PLOTTER — Vector plotter DT_RASDISPLAY — Raster display DT_RASPRINTER — Raster printer DT_RASCAMERA — Raster camera DT_CHARSTREAM — Character stream DT_METAFILE — Metafile DT_DISPFILE — Display file
HORZSIZE	Width of the physical display (in millimeters).
VERTSIZE	Height of the physical display (in millimeters).
HORZRES	Width of the display (in pixels).
VERTRES	Height of the display (in raster lines).
LOGPIXELSX	Number of pixels per logical inch along the display width.
LOGPIXELSY	Number of pixels per logical inch along the display height.
BITSPIXEL	Number of adjacent color bits for each pixel.
PLANES	Number of color planes.
NUMBRUSHES	Number of device-specific brushes.
NUMPENS	Number of device-specific pens.
NUMFONTS	Number of device-specific fonts.
NUMCOLORS	Number of entries in the device's color table.
ASPECTX	Relative width of a device pixel as used for line drawing.
ASPECTY	Relative height of a device pixel as used for line drawing.

Table 4.9: GDI information indexes (continued)

ASPECTXY	Diagonal width of the device pixel as used for line drawing.
PDEVICESIZE	Size of the **PDEVICE** internal data structure.
CLIPCAPS	Flag that indicates the clipping capabilities of the device. It is 1 if the device can clip to a rectangle, 0 if it cannot.
SIZEPALETTE	Number of entries in the system palette. This index is valid only if the device driver sets the RC_PALETTE bit in the RASTERCAPS index and is available only if the driver version is 3.0 or higher.
NUMRESERVED	Number of reserved entries in the system palette. This index is valid only if the device driver sets the RC_PALETTE bit in the RASTERCAPS index and is available only if the driver version is 3.0 or higher.
COLORRES	Actual color resolution of the device in bits per pixel. This index is valid only if the device driver sets the RC_PALETTE bit in the RASTERCAPS index and is available only if the driver version is 3.0 or higher.
RASTERCAPS	Value that indicates the raster capabilities of the device, as shown in the following list:

Capability	Meaning
RC_BANDING	Requires banding support.
RC_BITBLT	Capable of transferring bitmaps.
RC_BITMAP64	Capable of supporting bitmaps larger than 64K.
RC_DI_BITMAP	Capable of supporting **SetDIBits** and **GetDIBits**.
RC_DIBTODEV	Capable of supporting the **SetDIBitsToDevice** function.
RC_FLOODFILL	Capable of performing flood fills.
RC_GDI20_OUTPUT	Capable of supporting Windows version 2.0 features.
RC_PALETTE	Palette-based device.
RC_SCALING	Capable of scaling.
RC_STRETCHBLT	Capable of performing the **StretchBlt** function.
RC_STRETCHDIB	Capable of performing the **StretchDIBits** function.

CURVECAPS	A bitmask that indicates the curve capabilities of the device. The bits have the following meanings:

Bit	Meaning
0	Device can do circles.
1	Device can do pie wedges.
2	Device can do chord arcs.
3	Device can do ellipses.
4	Device can do wide borders.
5	Device can do styled borders.
6	Device can do borders that are wide and styled.
7	Device can do interiors.

Table 4.9: GDI information indexes (continued)

	The high byte is 0.
LINECAPS	A bitmask that indicates the line capabilities of the device. The bits have the following meanings: **Bit** — **Meaning** 0 — Reserved. 1 — Device can do polyline. 2 — Reserved. 3 — Reserved. 4 — Device can do wide lines. 5 — Device can do styled lines. 6 — Device can do lines that are wide and styled. 7 — Device can do interiors. The high byte is 0.
POLYGONALCAPS	A bitmask that indicates the polygonal capabilities of the device. The bits have the following meanings: **Bit** — **Meaning** 0 — Device can do alternate fill polygon. 1 — Device can do rectangle. 2 — Device can do winding number fill polygon. 3 — Device can do scanline. 4 — Device can do wide borders. 5 — Device can do styled borders. 6 — Device can do borders that are wide and styled. 7 — Device can do interiors. The high byte is 0.
TEXTCAPS	A bitmask that indicates the text capabilities of the device. The bits have the following meanings: **Bit** — **Meaning** 0 — Device can do character output precision. 1 — Device can do stroke output precision. 2 — Device can do stroke clip precision. 3 — Device can do 90-degree character rotation. 4 — Device can do any character rotation. 5 — Device can do scaling independent of *X* and *Y*. 6 — Device can do doubled character for scaling.

Table 4.9: GDI information indexes (continued)

7	Device can do integer multiples for scaling.
8	Device can do any multiples for exact scaling.
9	Device can do double-weight characters.
10	Device can do italicizing.
11	Device can do underlining.
12	Device can do strikeouts.
13	Device can do raster fonts.
14	Device can do vector fonts.
15	Reserved. Must be returned zero.

For a list of all the available abilities, see the **LOGFONT** data structure in Chapter 7, "Data types and structures," in *Reference, Volume 2*.

GetDialogBaseUnits 3.0

Syntax LONG GetDialogBaseUnits( )
function GetDialogBaseUnits: Longint;

This function returns the dialog base units used by Windows when creating dialog boxes. An application should use these values to calculate the average width of characters in the system font.

Parameters None.

Return value The return value specifies the dialog base units. The high-order word contains the height in pixels of the current dialog base height unit derived from the height of the system font, and the low-order word contains the width in pixels of the current dialog base width unit derived from the width of the system font.

Comments The values returned represent dialog base units before being scaled to actual dialog units. The actual dialog unit in the *x* direction is 1/4 of the width returned by **GetDialogBaseUnits**. The actual dialog unit in the *y* direction is 1/8 of the height returned by the function.

To determine the actual height and width in pixels of a control, given the height (x) and width (y) in dialog units and the return value (lDlgBaseUnits) from calling **GetDialogBaseUnits**, use the following formula:

```
(x * LOWORD(lDlgBaseUnits))/4
(y * HIWORD(lDlgBaseUnits))/8
```

To avoid rounding problems, perform the multiplication before the division in case the dialog base units are not evenly divisible by four.

GetDIBits 3.0

Syntax int GetDIBits(hDC, hBitmap, nStartScan, nNumScans, lpBits, lpBitsInfo, wUsage)
function GetDIBits(DC: HDC; Bitmap: THandle; StartScan, NumScans: Word; Bits: Pointer; var BitInfo: TBitmapInfo; Usage: Word): Integer;

This function retrieves the bits of the specified bitmap and copies them, in device-independent format, into the buffer that is pointed to by the *lpBits* parameter. The *lpBitsInfo* parameter retrieves the color format for the device-independent bits.

Parameters

hDC	**HDC** Identifies the device context.
hBitmap	**HBITMAP** Identifies the bitmap.
nStartScan	**WORD** Specifies the first scan line in the destination bitmap to set in *lpBits*.
nNumScans	**WORD** Specifies the number of lines to be copied.
lpBits	**LPSTR** Points to a buffer that will receive the bitmap bits in device-independent format.
lpBitsInfo	**LPBITMAPINFO** Points to a **BITMAPINFO** data structure that specifies the color format and dimension for the device-independent bitmap.
wUsage	**WORD** Specifies whether the **bmiColors[ ]** fields of the *lpBits-Info* parameter are to contain explicit RGB values or indexes into the currently realized logical palette. The *wUsage* parameter must be one of the following values:

Value	Meaning
DIB_PAL_COLORS	The color table is to consist of an array of 16-bit indexes into the currently realized logical palette.
DIB_RGB_COLORS	The color table is to contain literal RGB values.

Return value The return value specifies the number of scan lines copied from the bitmap. It is zero if there was an error.

Comments If the *lpBits* parameter is NULL, **GetDIBits** fills in the **BITMAPINFO** data structure to which the *lpBitsInfo* parameter points, but does not retrieve bits from the bitmap.
The bitmap identified by the *hBitmap* parameter must not be selected into a device context when the application calls this function.

The origin for device-independent bitmaps is the bottom-left corner of the bitmap, not the top-left corner, which is the origin when the mapping mode is MM_TEXT.

This function also retrieves a bitmap specification formatted for Microsoft OS/2 Presentation Manager versions 1.1 and 1.2 if the *lpBitsInfo* parameter points to a **BITMAPCOREINFO** data structure.

GetDlgCtrlID 3.0

Syntax int GetDlgCtrlID(hWnd)
function GetDlgCtrlID(Wnd: HWnd): Integer;

This function returns the ID value of the child window identified by the *hWnd* parameter.

Parameters *hWnd*, **HWND** Identifies the child window.

Return value The return value is the numeric identifier of the child window if the function is successful. If the function fails, or if *hWnd* is not a valid window handle, the return value is NULL.

Comments Since top-level windows do not have an ID value, the return value of this function is invalid if the *hWnd* parameter identifies a top-level window.

GetDlgItem

Syntax HWND GetDlgItem(hDlg, nIDDlgItem)
function GetDlgItem(Dlg: HWnd; IDDlgItem: Integer): HWnd;

This function retrieves the handle of a control contained in the dialog box specified by the *hDlg* parameter.

Parameters

hDlg	**HWND** Identifies the dialog box that contains the control.
nIDDlgItem	**int** Specifies the integer ID of the item to be retrieved.

Return value The return value identifies the given control. It is NULL if no control with the integer ID given by the *nIDDlgItem* parameter exists.

Comments The **GetDlgItem** function can be used with any parent-child window pair, not just dialog boxes. As long as the *hDlg* parameter specifies a parent window and the child window has a unique ID (as specified by the *hMenu* parameter in the **CreateWindow** function that created the child window), **GetDlgItem** returns a valid handle to the child window.

GetDlgItemInt

Syntax WORD GetDlgItemInt(hDlg, nIDDlgItem, lpTranslated, bSigned)
function GetDlgItemInt(Dlg: HWnd; IDDlgItem: Integer; Translate: PBool; Signed: Bool): Word;

This function translates the text of a control in the given dialog box into an integer value. The **GetDlgItemInt** function retrieves the text of the control identified by the *nIDDlgItem* parameter. It translates the text by stripping any extra spaces at the beginning of the text and converting decimal digits, stopping the translation when it reaches the end of the text or encounters any nonnumeric character. If the *bSigned* parameter is nonzero, **GetDlgItemInt** checks for a minus sign (–) at the beginning of the text and translates the text into a signed number. Otherwise, it creates an unsigned value.

GetDlgItemInt returns zero if the translated number is greater than 32,767 (for signed numbers) or 65,535 (for unsigned). When errors occur, such as encountering nonnumeric characters and exceeding the given maximum, **GetDlgItemInt** copies zero to the location pointed to by the *lpTranslated* parameter. If there are no errors, *lpTranslated* receives a nonzero value. If *lpTranslated* is NULL, **GetDlgItemInt** does not warn about errors. **GetDlgItemInt** sends a WM_GETTEXT message to the control.

Parameters

hDlg	**HWND** Identifies the dialog box.
nIDDlgItem	**int** Specifies the integer identifier of the dialog-box item to be translated.
lpTranslated	**BOOL FAR *** Points to the Boolean variable that is to receive the translated flag.
bSigned	**BOOL** Specifies whether the value to be retrieved is signed.

Return value The return value specifies the translated value of the dialog-box item text. Since zero is a valid return value, the *lpTranslated* parameter must be used to detect errors. If a signed return value is desired, it should be cast as an **int** type.

GetDlgItemText

Syntax int GetDlgItemText(hDlg, nIDDlgItem, lpString, nMaxCount)
function GetDlgItemText(Dlg: HWnd; IDDlgItem: Integer; Str: PChar; MaxCount: Integer): Integer;

This function retrieves the caption or text associated with a control in a dialog box. The **GetDlgItemText** function copies the text to the location pointed to by the *lpString* parameter and returns a count of the number of characters it copies.

GetDlgItemText sends a WM_GETTEXT message to the control.

Parameters

hDlg	**HWND** Identifies the dialog box that contains the control.
nIDDlgItem	**int** Specifies the integer identifier of the dialog-box item whose caption or text is to be retrieved.
lpString	**LPSTR** Points to the buffer to receive the text.
nMaxCount	**int** Specifies the maximum length (in bytes) of the string to be copied to *lpString*. If the string is longer than *nMaxCount*, it is truncated.

Return value The return value specifies the actual number of characters copied to the buffer. It is zero if no text is copied.

GetDOSEnvironment 3.0

Syntax LPSTR GetDOSEnvironment()
function GetDOSEnvironment: PChar;

This function returns a far pointer to the environment string of the currently running task. See a DOS technical reference manual for more information on the format and contents of the environment string.

Parameters None.

Comments Unlike an application, a dynamic-link library (DLL) does not have a copy of the environment string. As a result, a library must call this function to retrieve the environment string.

GetDoubleClickTime

Syntax WORD GetDoubleClickTime( )
function GetDoubleClickTime: Word;

This function retrieves the current double-click time for the mouse. A double-click is a series of two clicks of the mouse button, the second occurring within a specified time after the first. The double-click time is the maximum number of milliseconds that may occur between the first and second click of a double-click.

Parameters None.

Return value The return value specifies the current double-click time (in milliseconds).

GetDriveType 3.0

Syntax WORD GetDriveType(nDrive)
function GetDriveType(Drive: Integer): Word;

This function determines whether a disk drive is removeable, fixed, or remote.

Parameters *nDrive* **int** Specifies the drive for which the type is to be determined. Drive A: is 0, drive B: is 1, drive C: is 2, and so on.

Return value The return value specifies the type of drive. It can be one of the following values:

Value	Meaning
DRIVE_REMOVEABLE	Disk can be removed from the drive.
DRIVE_FIXED	Disk cannot be removed from the drive.
DRIVE_REMOTE	Drive is a remote (network) drive.

The return value is zero if the function cannot determine the drive type, or 1 if the specified drive does not exist.

GetEnvironment

Syntax int GetEnvironment(lpPortName, lpEnviron, nMaxCount)
function GetEnvironment(PortName: PChar; Environ: Pointer; MaxCount: Word): Integer;

This function retrieves the current environment that is associated with the device attached to the system port specified by the *lpPortName* parameter, and copies it into the buffer specified by the *lpEnviron* parameter. The environment, maintained by GDI, contains binary data used by GDI whenever a device context is created for the device on the given port.

The function fails if there is no environment for the given port.

An application can call this function with the *lpEnviron* parameter set to NULL to determine the size of the buffer required to hold the environment. It can then allocate the buffer and call **GetEnvironment** a second time to retrieve the environment.

Parameters

lpPortName **LPSTR** Points to the null-terminated character string that specifies the name of the desired port.

lpEnviron **LPSTR** Points to the buffer that will receive the environment.

nMaxCount **WORD** Specifies the maximum number of bytes to copy to the buffer.

Return value The return value specifies the number of bytes copied to *lpEnviron*. If *lpEnviron* is NULL, the return value is the size in bytes of the buffer required to hold the environment. It is zero if the environment cannot be found.

Comments The first field in the buffer pointed to by *lpEnviron* must be the same as that passed in the *lpDeviceName* parameter of the **CreateDC** function. If *lpPortName* specifies a null port (as defined in the WIN.INI file), the device name pointed to by *lpEnviron* is used to locate the desired environment.

GetFocus

Syntax

```
HWND GetFocus( )
function GetFocus: HWnd;
```

This function retrieves the handle of the window that currently owns the input focus.

Parameters None.

Return value The return value identifies the window that currently owns the focus if the function is successful. Otherwise, it is NULL.

GetFreeSpace 3.0

Syntax DWORD GetFreeSpace(wFlags)
function GetFreeSpace(Flag: Word): Longint;

This function scans the global heap and returns the number of bytes of memory currently available.

Parameters *wFlags* **WORD** Specifies whether to scan the heap above or below the EMS bank line in large-frame and small-frame EMS systems. If it is set to GMEM_NOT_BANKED, **GetFreeSpace** returns the amount of memory available below the line. If *wFlags* is zero, **GetFreeSpace** returns the amount is the memory available above the EMS bank line. The *wFlags* parameter is ignored for non-EMS systems.

Return value The return value is the amount of available memory in bytes. This memory is not necessarily contiguous; the **GlobalCompact** function returns the number of bytes in the largest block of free global memory.

Comments In standard mode, the value returned represents the number of bytes in the global heap that are not used and that are not reserved for code. In 386 enhanced mode, the value returned is calculated using the following formula:

Free_space = (*heap* – *reserved*) + (*page_file* + *phys_pages*) – (*total_linear* – *free_linear*) – 64K

In this formula:

- *heap* is the number of unused bytes in the global heap.
- *reserved* is the number of unused bytes in the global heap reserved for code.
- *page_file* is the size of the paging file.
- *phys_page* is the total size of physical pages.
- *total_linear* is the total linear address space.
- *free_linear* is the total unused linear address space

The return value in 386 enhanced mode is an estimate of the amount of memory available to an application. It does not account for memory held in reserve for non-Windows applications.

GetGValue

Syntax BYTE GetGValue(rgbColor)
function GetGValue(RGBColor: Longint): Byte;

This macro extracts the green value from an RGB color value.

Parameters *rgbColor* **DWORD** Specifies a red, a green, and a blue color field, each specifying the intensity of the given color.

Return value The return value specifies a byte that contains the green value of the *rgbColor* parameter.

Comments The value 0FFH corresponds to the maximum intensity value for a single byte; 000H corresponds to the minimum intensity value for a single byte.

GetInputState

Syntax BOOL GetInputState( )
function GetInputState: Bool;

This function determines whether there are mouse, keyboard, or timer events in the system queue that require processing. An event is a record that describes interrupt-level input. Mouse events occur when a user moves the mouse or clicks a mouse button. Keyboard events occur when a user presses one or more keys. Timer events occur after a specified number of clock ticks. The system queue is the location in which Windows stores mouse, keyboard, and timer events.

Parameters None.

Return value The return value specifies whether mouse, keyboard or timer input occurs. It is nonzero if input is detected. Otherwise, it is zero.

GetInstanceData

Syntax int GetInstanceData(hInstance, pData, nCount)
function GetInstanceData(Instance: THandle; Data: Word; Count: Integer): Integer;

This function copies data from a previous instance of an application into the data area of the current instance. The *hInstance* parameter specifies which instance to copy data from, *pData* specifies where to copy the data, and *nCount* specifies the number of bytes to copy.

Parameters	*hInstance*	**HANDLE** Identifies a previous call of the application.
	pData	**NPSTR** Points to a buffer in the current instance.
	nCount	**int** Specifies the number of bytes to copy.

Return value The return value specifies the number of bytes actually copied.

GetKBCodePage 3.0

Syntax int GetKBCodePage()
function GetKBCodePage: Integer;

This function determines which OEM/ANSI tables are loaded by Windows.

Parameters None.

Return value The return value specifies the code page currently loaded by Windows. It can be one of the following values:

Value	Meaning
437	Default (USA, used by most countries: indicates that there is no OEMANSI.BIN in the Windows directory)
850	International (OEMANSI.BIN = XLAT850.BIN)
860	Portugal (OEMANSI.BIN = XLAT860.BIN)
861	Iceland (OEMANSI.BIN = XLAT861.BIN)
863	French Canadian (OEMANSI.BIN = XLAT863.BIN)
865	Norway/Denmark (OEMANSI.BIN = XLAT865.BIN)

Comments If the file OEMANSI.BIN is in the Windows directory, Windows reads it and overwrites the OEM/ANSI translation tables in the keyboard driver.

When the user selects a language within the Setup program and the language does not use the default code page (437), Setup copies the appropriate file (such as XLATPO.BIN) to OEMANSI.BIN in the Windows system directory. If the language uses the default code page, Setup deletes OEMANSI.BIN, if it exists, from the Windows system directory.

GetKeyboardState

Syntax void GetKeyboardState(lpKeyState)
procedure GetKeyboardState(var KeyState: TKeyboardState);

This function copies the status of the 256 virtual-keyboard keys to the buffer specified by the *lpKeyState* parameter. The high bit of each byte is

set to 1 if the key is down, or it is set to 0 if it is up. The low bit is set to 1 if the key was pressed an odd number of times since startup. Otherwise, it is set to 0.

Parameters *lpKeyState* **BYTE FAR** * Points to the 256-byte buffer of virtual-key codes.

Return value None.

Comments An application calls the **GetKeyboardState** function in response to a keyboard-input message. This function retrieves the state of the keyboard when the input message was generated.

To obtain state information for individual keys, follow these steps:

1. Create an array of characters that is 265 bytes long.
2. Copy the contents of the buffer pointed to by the *lpKeyState* parameter into the array.
3. Use the virtual-key code from Appendix A, "Virtual-key codes," in *Reference, Volume 2*, to obtain an individual key state.

GetKeyboardType 3.0

Syntax int GetKeyboardType(nTypeFlag)
function GetKeyboardType(TypeFlag: Integer): Integer;

This function retrieves the system-keyboard type.

Parameters *nTypeFlag*, **int** Determines whether the function returns a value indicating the type or subtype of the keyboard. It may be one of the following values:

Value	Meaning
0	Function returns the keyboard type.
1	Function returns the keyboard subtype.
2	Function returns the number of function keys on the keyboard.

Return value The return value indicates the type or subtype of the system keyboard or the number of function keys on the keyboard. The subtype is an OEM-dependent value. The type may be one of the following values:

Value	Meaning
1	IBM® PC/XT®, or compatible (83-key) keyboard
2	Olivetti® M24 "ICO" (102-key) keyboard
3	IBM AT® (84-key) or similar keyboard
4	IBM Enhanced (101- or 102-key) keyboard
5	Nokia 1050 and similar keyboards
6	Nokia 9140 and similar keyboards

The return value is zero if the *nTypeFlag* parameter is greater than 2 or if the function fails.

Comments An application can determine the number of function keys on a keyboard from the keyboard type. The following shows the number of function keys for each keyboard type:

Type	Number of Function Keys
1	10
2	12 (sometimes 18)
3	10
4	12
5	10
6	24

GetKeyNameText 3.0

Syntax int GetKeyNameText(lParam, lpBuffer, nSize)
function GetKeyNameText(lParam: Longint; Buffer: PChar; Size: Integer): Integer;

This function retrieves a string which contains the name of a key.

The keyboard driver maintains a list of names in the form of character strings for keys with names longer than a single character. The key name is translated according to the layout of the currently installed keyboard. The translation is performed for the principal language supported by the keyboard driver.

Parameters

lParam **DWORD** Specifies the 32-bit parameter of the keyboard message (such as WM_KEYDOWN) which the function is processing. Byte 3 (bits 16–23) of the long parameter is a scan code. Bit 20 is the extended bit that distinguishes some keys on an enhanced keyboard. Bit 21 is a "don't care" bit; the application calling this function sets this bit to indicate that the function should not distinguish between left and right control and shift keys, for example.

lpBuffer	**LPSTR** Specifies a buffer to receive the key name.
nSize	**WORD** Specifies the maximum length in bytes of the key name, not including the terminating NULL character.

Return value The return value is the actual length of the string copied to *lpBuffer*.

GetKeyState

Syntax int GetKeyState(nVirtKey)
function GetKeyState(VirtKey: Integer): Integer;

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

Parameters

nVirtKey	**int** Specifies a virtual key. If the desired virtual key is a letter or digit (A through Z, a through z, or 0 through 9), *nVirtKey* must be set to the ASCII value of that character. For other keys, it must be one of the values listed in Appendix A, "Virtual-key codes," in *Reference, Volume 2*.

Return value The return value specifies the state of the given virtual key. If the high-order bit is 1, the key is down. Otherwise, it is up. If the low-order bit is 1, the key is toggled. A toggle key, such as the CAPSLOCK key, is toggled if it has been pressed an odd number of times since the system was started. The key is untoggled if the low bit is 0.

Comments An application calls the **GetKeyState** function in response to a keyboard-input message. This function retrieves the state of the key when the input message was generated.

GetLastActivePopup 3.0

Syntax HWND GetLastActivePopup(hwndOwner)
function GetLastActivePopup(Owner: HWnd): HWnd;

This function determines which pop-up window owned by the window identified by the *hwndOwner* parameter was most recently active.

Parameters

hwndOwner	**HWND** Identifies the owner window.

Return value The return value identifies the most-recently active pop-up window. The return value will be *hwndOwner* if any of the following conditions are met:

- The window identified by *hwndOwner* itself was most recently active.

- The window identified by *hwndOwner* does not own any pop-up windows.
- The window identified by *hwndOwner* is not a top-level window or is owned by another window.

GetMapMode

Syntax int GetMapMode(hDC)
function GetMapMode(DC: HDC): Integer;

This function retrieves the current mapping mode. See the **SetMapMode** function, later in this chapter, for a description of the mapping modes.

Parameters *hDC* **HDC** Identifies the device context.

Return value The return value specifies the mapping mode.

GetMenu

Syntax HMENU GetMenu(hWnd)
function GetMenu(Wnd: HWnd): HMenu;

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

Parameters *hWnd* **HWND** Identifies the window whose menu is to be examined.

Return value The return value identifies the menu. It is NULL if the given window has no menu. The return value is undefined if the window is a child window.

GetMenuCheckMarkDimensions 3.0

Syntax DWORD GetMenuCheckMarkDimensions( )
function GetMenuCheckMarkDimensions: Longint;

This function returns the dimensions of the default checkmark bitmap. Windows displays this bitmap next to checked menu items. Before calling the **SetMenuItemBitmaps** function to replace the default checkmark, an application should call the **GetMenuCheckMarkDimensions** function to determine the correct size for the bitmaps.

Parameters None.

Return value The return value specifies the height and width of the default checkmark bitmap. The high-order word contains the height in pixels and the low-order word contains the width.

GetMenuItemCount

Syntax WORD GetMenuItemCount(hMenu)
function GetMenuItemCount(Menu: HMenu): Word;

This function determines the number of items in the menu identified by the *hMenu* parameter. This may be either a pop-up or a top-level menu.

Parameters *hMenu* **HMENU** Identifies the handle to the menu to be examined.

Return value The return value specifies the number of items in the menu specified by the *hMenu* parameter if the function is successful. Otherwise, it is –1.

GetMenuItemID

Syntax WORD GetMenuItemID(hMenu, nPos)
function GetMenuItemID(Menu: HMenu; Pos: Integer): Word;

This function obtains the menu-item identifier for a menu item located at the position defined by the *nPos* parameter.

Parameters

hMenu	**HMENU** Identifies a handle to the pop-up menu that contains the item whose ID is being retrieved.
nPos	**int** Specifies the position (zero-based) of the menu item whose ID is being retrieved.

Return value The return value specifies the item ID for the specified item in a pop-up menu if the function is successful; if *hMenu* is NULL or if the specified item is a pop-up menu (as opposed to an item within the pop-up menu), the return value is –1.

GetMenuState

Syntax WORD GetMenuState(hMenu, wId, wFlags)
function GetMenuState(Menu: HMenu; ID, Flags: Word): Word;

This function obtains the number of items in the pop-up menu associated with the menu item specified by the *wId* parameter if the *hMenu*

parameter identifies a menu with an associated pop-up menu. If *hMenu* identifies a pop-up menu, this function obtains the status of the menu item associated with *wId*.

Parameters

hMenu	**HMENU** Identifies the menu.
wId	**WORD** Specifies the menu-item ID.
wFlags	**WORD** Specifies the nature of the *wId* parameter. If the *wFlags* parameter contains MF_BYPOSITION, *wId* specifies a (zero-based) relative position; if *wFlags* contains MF_BYCOMMAND, *wId* specifies the item ID.

Return value The return value specifies the outcome of the function. It is –1 if the specified item does not exist. If the menu itself does not exist, a fatal exit occurs. If *wId* identifies a pop-up menu, the return value contains the number of items in the pop-up menu in its high-order byte, and the menu flags associated with the pop-up menu in its low-order byte; otherwise, it is a mask (Boolean OR) of the values from the following list (this mask describes the status of the menu item that *wId* identifies):

Value	Meaning
MF_CHECKED	Checkmark is placed next to item (pop-up menus only).
MF_DISABLED	Item is disabled.
MF_ENABLED	Item is enabled.
MF_GRAYED	Item is disabled and grayed.
MF_MENUBARBREAK	Same as MF_MENUBREAK, except for pop-up menus where the new column is separated from the old column by a vertical dividing line.
MF_MENUBREAK	Item is placed on a new line (static menus) or in a new column (pop-up menus) without separating columns.
MF_SEPARATOR	Horizontal dividing line is drawn (pop-up menus only). This line cannot be enabled, checked, grayed, or highlighted. The *lpNewItem* and *wIDNewItem* parameters are ignored.
MF_UNCHECKED	Checkmark is not placed next to item (default).

GetMenuString

Syntax int GetMenuString(hMenu, wIDItem, lpString, nMaxCount, wFlag)
function GetMenuString(Menu: HMenu; IDItem: Word; Str: PChar; MaxCount: Integer; Flag: Word): Integer;

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

Parameters

hMenu	**HMENU** Identifies the menu.

wIDItem	**WORD** Specifies the integer identifier of the menu item (from the resource file) or the offset of the menu item in the menu, depending on the value of the *wFlag* parameter.
lpString	**LPSTR** Points to the buffer that is to receive the label.
nMaxCount	**int** Specifies the maximum length of the label to be copied. If the label is longer than the maximum specified in *nMaxCount*, the extra characters are truncated.
wFlag	**WORD** Specifies the nature of the *wID* parameter. If *wFlags* contains MF_BYPOSITION, *wId* specifies a (zero-based) relative position; if the *wFlags* parameter contains MF_BYCOMMAND, *wId* specifies the item ID.

Return value The return value specifies the actual number of bytes copied to the buffer.

Comments The *nMaxCount* parameter should be one larger than the number of characters in the label to accommodate the null character that terminates a string.

GetMessage

Syntax BOOL GetMessage(lpMsg, hWnd, wMsgFilterMin, wMsgFilterMax)
function GetMessage(var Msg: TMsg; Wnd: HWnd; MsgFilterMin, MsgFilterMax: Word): Bool;

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

GetMessage retrieves only messages associated with the window specified by the *hWnd* parameter and within the range of message values given by the *wMsgFilterMin* and *wMsgFilterMax* parameters. If *hWnd* is NULL, **GetMessage** retrieves messages for any window that belongs to the application making the call. (The **GetMessage** function does not retrieve messages for windows that belong to other applications.) If *wMsgFilterMin* and *wMsgFilterMax* are both zero, **GetMessage** returns all available messages (no filtering is performed).

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

G

Parameters

lpMsg	**LPMSG** Points to an **MSG** data structure that contains message information from the Windows application queue.
hWnd	**HWND** Identifies the window whose messages are to be examined. If *hWnd* is NULL, **GetMessage** retrieves messages for any window that belongs to the application making the call.
wMsgFilterMin	**WORD** Specifies the integer value of the lowest message value to be retrieved.
wMsgFilterMax	**WORD** Specifies the integer value of the highest message value to be retrieved.

Return value The return value specifies the outcome of the function. It is nonzero if a message other than WM_QUIT is retrieved. It is zero if the WM_QUIT message is retrieved.

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

Comments In addition to yielding control to other applications when no messages are available, the **GetMessage** and **PeekMessage** functions also yield control when WM_PAINT or WM_TIMER messages for other tasks are available.

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

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

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

GetMessagePos

Syntax DWORD GetMessagePos( )
function GetMessagePos: Longint;

This function returns a long value that represents the cursor position (in screen coordinates) when the last message obtained by the **GetMessage** function occurred.

Parameters None.

Return value The return value specifies the *x*- and *y*-coordinates of the cursor position. The *x*-coordinate is in the low-order word, and the *y*-coordinate is in the high-order word. If the return value is assigned to a variable, the **MAKEPOINT** macro can be used to obtain a **POINT** structure from the return value; the **LOWORD** or **HIWORD** macro can be used to extract the *x*- or the *y*-coordinate.

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

GetMessageTime

Syntax DWORD GetMessageTime( )
function GetMessageTime: Longint;

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

Parameters None.

Return value The return value specifies the message time.

Comments Do not assume that the return value is always increasing. The return value will "wrap around" to zero if the timer count exceeds the maximum value for long integers.

To calculate time delays between messages, subtract the time of the second message from the time of the first message.

GetMetaFile

Syntax HANDLE GetMetaFile(lpFilename)
function GetMetaFile(FileName: PChar): THandle;

This function creates a handle for the metafile named by the *lpFilename* parameter.

Parameters *lpFilename* **LPSTR** Points to the null-terminated character string that specifies the DOS filename of the metafile. The metafile is assumed to exist.

Return value The return value identifies a metafile if the function is successful. Otherwise, it is NULL.

GetMetaFileBits

Syntax HANDLE GetMetaFileBits(hMF)
function GetMetaFileBits(MF: THandle): THandle;

This function returns a handle to a global memory block that contains the specified metafile as a collection of bits. The memory block can be used to determine the size of the metafile or to save the metafile as a file. The memory block should not be modified.

Parameters *hMF* **HANDLE** Identifies the memory metafile.

Return value The return value identifies the global memory block that contains the metafile. If an error occurs, the return value is NULL.

Comments The handle used as the *hMF* parameter becomes invalid when the **GetMetaFileBits** function returns, so the returned global memory handle must be used to refer to the metafile.

Memory blocks created by this function are unique to the calling application and are not shared by other applications. These blocks are automatically deleted when the application terminates.

GetModuleFileName

Syntax int GetModuleFileName(hModule, lpFilename, nSize)

function GetModuleFileName(Module: THandle; FileName: PChar; Size: Integer): Integer;

This function retrieves the full pathname of the executable file from which the specified module was loaded. The function copies the null-terminated filename into the buffer pointed to by the *lpFilename* parameter.

Parameters

hModule	**HANDLE** Identifies the module or the instance of the module.
lpFilename	**LPSTR** Points to the buffer that is to receive the filename.
nSize	**int** Specifies the maximum number of characters to copy. If the filename is longer than the maximum number of characters specified by the *nSize* parameter, it is truncated.

Return value The return value specifies the actual length of the string copied to the buffer.

GetModuleHandle

Syntax HANDLE GetModuleHandle(lpModuleName)
function GetModuleHandle(ModuleName: PChar): THandle;

This function retrieves the module handle of the specified module.

Parameters

lpModuleName	**LPSTR** Points to a null-terminated character string that specifies the module.

Return value The return value identifies the module if the function is successful. Otherwise, it is NULL.

GetModuleUsage

Syntax int GetModuleUsage(hModule)
function GetModuleUsage(Module: THandle): Integer;

This function returns the reference count of a specified module.

Parameters

hModule	**HANDLE** Identifies the module or an instance of the module.

Return value The return value specifies the reference count of the module.

GetNearestColor

Syntax DWORD GetNearestColor(hDC, crColor)
function GetNearestColor(DC: HDC; Color: TColorRef): TColorRef;

This function returns the closest logical color to a specified logical color the given device can represent.

Parameters

hDC	**HDC** Identifies the device context.
crColor	**COLORREF** Specifies the color to be matched.

Return value The return value specifies an RGB color value that names the solid color closest to the *crColor* value that the device can represent.

GetNearestPaletteIndex 3.0

Syntax WORD GetNearestPaletteIndex(hPalette, crColor)
function GetNearestPaletteIndex(Palette: HPalette; Color: TColorRef): Word;

This function returns the index of the entry in a logical palette which most closely matches a color value.

Parameters

hPalette	**HPALETTE** Identifies the logical palette.
crColor	**COLORREF** Specifies the color to be matched.

Return value The return value is the index of an entry in a logical palette. The entry contains the color which most nearly matches the specified color.

GetNextDlgGroupItem

Syntax HWND GetNextDlgGroupItem(hDlg, hCtl, bPrevious)
function GetNextDlgGroupItem(Dlg, Ctrl: HWnd; Previous: Bool): HWnd;

This function searches for the next (or previous) control within a group of controls in the dialog box identified by the *hDlg* parameter. A group of controls consists of one or more controls with WS_GROUP style.

Parameters

hDlg	**HWND** Identifies the dialog box being searched.

hCtl	**HWND** Identifies the control in the dialog box where the search starts.
bPrevious	**BOOL** Specifies how the function is to search the group of controls in the dialog box. If the *bPrevious* parameter is zero, the function searches for the previous control in the group. If *bPrevious* is nonzero, the function searches for the next control in the group.

Return value The return value identifies the next or previous control in the group.

Comments If the current item is the last item in the group and *bPrevious* is zero, the **GetNextDlgGroupItem** function returns the window handle of the first item in the group. If the current item is the first item in the group and *bPrevious* is nonzero, **GetNextDlgGroupItem** returns the window handle of the last item in the group.

GetNextDlgTabItem

Syntax HWND GetNextDlgTabItem(hDlg, hCtl, bPrevious)
function GetNextDlgTabItem(Dlg, Ctrl: HWnd; Previous: Bool): HWnd;

This function obtains the handle of the first control that has the WS_TABSTOP style that precedes (or follows) the control identified by the *hCtl* parameter.

Parameters

hDlg	**HWND** Identifies the dialog box being searched.
hCtl	**HWND** Identifies the control to be used as a starting point for the search.
bPrevious	**BOOL** Specifies how the function is to search the dialog box. If the *bPrevious* parameter is zero, the function searches for the previous control in the dialog box. If *bPrevious* is nonzero, the function searches for the next control in the dialog box. Identifies the control to be used as a starting point for the search.

Return value The return value identifies the previous (or next) control that has the WS_TABSTOP style set.

GetNextWindow

Syntax HWND GetNextWindow(hWnd, wFlag)
function GetNextWindow(Wnd: HWnd; Flag: Word): HWnd;

This function searches for a handle that identifies the next (or previous) window in the window-manager's list. The window-manager's list contains entries for all top-level windows, their associated child windows, and the child windows of any child windows. If the *hWnd* parameter is a handle to a top-level window, the function searches for the next (or previous) handle to a top-level window; if *hWnd* is a handle to a child window, the function searches for a handle to the next (or previous) child window.

Parameters

hWnd — **HWND** Identifies the current window.

wFlag — **WORD** Specifies whether the function returns a handle to the next window or to the previous window. It can be either of the following values:

Value	Meaning
GW_HWNDNEXT	The function returns a handle to the next window.
GW_HWNDPREV	The function returns a handle to the previous window.

Return value The return value identifies the next (or the previous) window in the window-manager's list.

GetNumTasks

Syntax int GetNumTasks( )
function GetNumTasks: Word;

This function returns the number of tasks currently executing in the system. A task is a unique instance of a Windows application.

Parameters None.

Return value The return value specifies an integer that represents the number of tasks currently executing in the system.

GetObject

Syntax int GetObject(hObject, nCount, lpObject)
function GetObject(hObject: THandle; Count: Integer; lpObjectPtr: Pointer): Integer;

This function fills a buffer with the logical data that defines the logical object specified by the *hObject* parameter. The **GetObject** function copies

the number of bytes of data specified by the *nCount* parameter to the buffer pointed to by the *lpObject* parameter. The function retrieves data structures of the **LOGPEN, LOGBRUSH, LOGFONT**, or **BITMAP** type, or an integer, depending on the logical object. The buffer must be sufficiently large to receive the data.

If *hObject* specifies a bitmap, the function returns only the width, height, and color format information of the bitmap. The actual bits must be retrieved by using the **GetBitmapBits** function.

If *hObject* specifies a logical palette, it retrieves a two-byte value that specifies the number of entries in the palette; it does not retrieve the entire **LOGPALETTE** data structure that defines the palette. To get information on palette entries, an application must call the **GetPaletteEntries** function.

Parameters

hObject	**HANDLE** Identifies a logical pen, brush, font, bitmap, or palette.
nCount	**int** Specifies the number of bytes to be copied to the buffer.
lpObject	**LPSTR** Points to the buffer that is to receive the information.

Return value The return value specifies the actual number of bytes retrieved. It is zero if an error occurs.

GetPaletteEntries 3.0

Syntax WORD GetPaletteEntries(hPalette, wStartIndex, wNumEntries, lpPaletteEntries)
function GetPaletteEntries(Palette: HPalette; StartIndex, NumEntries: Word; var PaletteEntries): Word;

This function retrieves a range of palette entries in a logical palette.

Parameters

hPalette	**HPALETTE** Identifies the logical palette.
wStartIndex	**WORD** Specifies the first entry in the logical palette to be retrieved.
wNumEntries	**WORD** Specifies the number of entries in the logical palette to be retrieved.
lpPaletteEntries	**LPPALETTEENTRY** Points to an array of **PALETTEENTRY** data structures to receive the palette entries. The array must contain at least as many data structures as specified by the *wNumEntries* parameter.

Return value The return value is the number of entries retrieved from the logical palette. It is zero if the function failed.

GetParent

Syntax HWND GetParent(hWnd)
function GetParent(Wnd: HWnd): HWnd;

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

Parameters

hWnd	**HWND** Identifies the window whose parent window handle is to be retrieved.

Return value The return value identifies the parent window. It is NULL if the window has no parent window.

GetPixel

Syntax DWORD GetPixel(hDC, X, Y)
function GetPixel(DC: HDC; X, Y; Integer): TColorRef;

This function retrieves the RGB color value of the pixel at the point specified by the *X* and *Y* parameters. The point must be in the clipping region. If the point is not in the clipping region, the function is ignored.

Parameters

hDC	**HDC** Identifies the device context.
X	**int** Specifies the logical *x*-coordinate of the point to be examined.
Y	**int** Specifies the logical *y*-coordinate of the point to be examined.

Return value The return value specifies an RGB color value for the color of the given point. It is –1 if the coordinates do not specify a point in the clipping region.

Comments Not all devices support the **GetPixel** function. For more information, see the RC_BITBLT raster capability in the **GetDeviceCaps** function, earlier in this chapter.

GetPolyFillMode

Syntax int GetPolyFillMode(hDC)
function GetPolyFillMode(DC: HDC): Integer;

This function retrieves the current polygon-filling mode.

Parameters *hDC* **HDC** Identifies the device context.

Return value The return value specifies the polygon-filling mode. It can be any one of the following values:

Value	Meaning
ALTERNATE	Alternate mode
WINDING	Winding-number mode

For a description of these modes, see the **SetPolyFillMode** function, later in this chapter.

GetPriorityClipboardFormat 3.0

Syntax int GetPriorityClipboardFormat(lpPriorityList, nCount)
function GetPriorityClipboardFormat(var PriorityList; Count: Integer): Integer;

This function returns the first clipboard format in a list for which data exist in the clipboard.

Parameters

lpPriorityList **WORD FAR *** Points to an integer array that contains a list of clipboard formats in priority order. For a description of the data formats, see the **SetClipboardData** function later in this chapter.

nCount **int** Specifies the number of entries in *lpPriorityList*. This value must not be greater than the actual number of entries in the list.

Return value The return value is the highest priority clipboard format in the list for which data exist. If no data exist in the clipboard, this function returns NULL. If data exist in the clipboard which did not match any format in the list, the return value is –1.

GetPrivateProfileInt 3.0

Syntax WORD GetPrivateProfileInt(lpApplicationName, lpKeyName, nDefault, lpFileName)
function GetPrivateProfileInt(ApplicationName, KeyName: PChar; Default: Integer; FileName: PChar): Integer;

This function retrieves the value of an integer key from the specified initialization file. The function searches the file for a key that matches the name specified by the *lpKeyName* parameter under the application heading specified by the *lpApplicationName* parameter. An integer entry in the initialization file must have the following form:

```
[application name]

keyname = value
:
```

Parameters

lpApplicationName
LPSTR Points to the name of a Windows application that appears in the initialization file.

lpKeyName **LPSTR** Points to a key name that appears in the initialization file.

nDefault **int** Specifies the default value for the given key if the key cannot be found in the initialization file.

lpFileName **LPSTR** Points to a string that names the initialization file. If *lpFileName* does not contain a path to the file, Windows searches for the file in the Windows directory.

Return value The return value specifies the result of the function. The return value is zero if the value that corresponds to the specified key name is not an integer or if the integer is negative. If the value that corresponds to the key name consists of digits followed by nonnumeric characters, the function returns the value of the digits. For example, if the entry *KeyName=102abc* is accessed, the function returns 102. If the key is not found, this function returns the default value, *nDefault*.

Comments The **GetPrivateProfileInt** function is not case dependent, so the strings in *lpApplicationName* and *lpKeyName* may be in any combination of uppercase and lowercase letters.

GetPrivateProfileString 3.0

Syntax int GetPrivateProfileString(lpApplicationName, lpKeyName, lpDefault, lpReturnedString, nSize, lpFileName)
function GetPrivateProfileString(ApplicationName, KeyName, Default, ReturnedString: PChar; Size: Integer; FileName: PChar): Integer;

This function copies a character string from the specified initialization file into the buffer pointed to by the *lpReturnedString* parameter.

The function searches the file for a key that matches the name specified by the *lpKeyName* parameter under the application heading specified by the *lpApplicationName* parameter. If the key is found, the corresponding string is copied to the buffer. If the key does not exist, the default character string specified by the *lpDefault* parameter is copied. A string entry in the initialization file must have the following form:

```
[application name]
keyname = string
:
```

If *lpKeyName* is NULL, the **GetPrivateProfileString** function enumerates all key names associated with *lpApplicationName* by filling the location pointed to by *lpReturnedString* with a list of key names (not values). Each key name in the list is terminated with a null character.

Parameters

lpApplicationName	**LPSTR** Points to the name of a Windows application that appears in the initialization file.
lpKeyName	**LPSTR** Points to a key name that appears in the initialization file.
lpDefault	**LPSTR** Specifies the default value for the given key if the key cannot be found in the initialization file.
lpReturnedString	**LPSTR** Points to the buffer that receives the character string.
nSize	**int** Specifies the maximum number of characters (including the last null character) to be copied to the buffer.
lpFileName	**LPSTR** Points to a string that names the initialization file. If *lpFileName* does not contain a path to the file, Windows searches for the file in the Windows directory.

Return value The return value specifies the number of characters copied to the buffer identified by the *lpReturnedString* parameter, not including the terminating null character. If the buffer is not large enough to contain the entire string and *lpKeyName* is not NULL, the return value is equal to the length specified by the *nSize* parameter. If the buffer is not large enough to contain the entire string and *lpKeyName* is NULL, the return value is equal to the length specified by the *nSize* parameter minus 2.

Comments **GetPrivateProfileString** is not case dependent, so the strings in *lpApplicationName* and *lpKeyName* may be in any combination of uppercase and lowercase letters.

GetProcAddress

Syntax FARPROC GetProcAddress(hModule, lpProcName)
function GetProcAddress(Module: THandle; ProcName: PChar): TFarProc;

This function retrieves the memory address of the function whose name is pointed to by the *lpProcName* parameter. The **GetProcAddress** function searches for the function in the module specified by the *hModule* parameter, or in the current module if *hModule* is NULL. The function must be an exported function; the module's definition file must contain an appropriate **EXPORTS** line for the function.

Parameters

hModule	**HANDLE** Identifies the library module that contains the function.
lpProcName	**LPSTR** Points to the function name, or contains the ordinal value of the function. If it is an ordinal value, the value must be in the low-order word and zero must be in the high-order word. The string must be a null-terminated character string.

Return value The return value points to the function's entry point if the function is successful. Otherwise, it is NULL.

If the *lpProcName* parameter is an ordinal value and a function with the specified ordinal does not exist in the module, **GetProcAddress** can still return a non-NULL value. In cases where the function may not exist, specify the function by name rather than ordinal value.

Comments Only use **GetProcAddress** to retrieve addresses of exported functions that belong to library modules. The **MakeProcInstance** function can be used to access functions within different instances of the current module.

The spelling of the function name (pointed to by *lpProcName*) must be identical to the spelling as it appears in the source library's definition (.DEF) file. The function can be renamed in the definition file.

GetProfileInt

Syntax WORD GetProfileInt(lpAppName, lpKeyName, nDefault)
function GetProfileInt(AppName, KeyName: PChar; Default: Integer): Integer;

This function retrieves the value of an integer key from the Windows initialization file, WIN.INI. The function searches WIN.INI for a key that matches the name specified by the *lpKeyName* parameter under the application heading specified by the *lpAppName* parameter. An integer entry in WIN.INI must have the following form:

```
[application name]

keyname = value
:
```

Parameters

lpAppName **LPSTR** Points to the name of a Windows application that appears in the Windows initialization file.

lpKeyName **LPSTR** Points to a key name that appears in the Windows initialization file.

nDefault **int** Specifies the default value for the given key if the key cannot be found in the Windows initialization file.

Return value The return value specifies the result of the function. The return value is zero if the value that corresponds to the specified key name is not an integer or if the integer is negative. If the value that corresponds to the key name consists of digits followed by nonnumeric characters, the function returns the value of the digits. For example, if the entry *KeyName=102abc* is accessed, the function returns 102. If the key is not found, this function returns the default value, *nDefault*.

GetProfileString

Syntax int GetProfileString(lpAppName, lpKeyName, lpDefault, lpReturnedString, nSize)
function GetProfileString(AppName, KeyName, Default, ReturnedString: PChar; Size: Integer): Integer;

This function copies a character string from the Windows initialization file, WIN.INI, into the buffer pointed to by the *lpReturnedString* parameter. The function searches WIN.INI for a key that matches the name specified by the *lpKeyName* parameter under the application heading specified by the *lpAppName* parameter. If the key is found, the corresponding string is copied to the buffer. If the key does not exist, the default character string specified by the *lpDefault* parameter is copied. A string entry in WIN.INI must have the following form:

```
[application name]

keyname = value
:
```

If *lpKeyName* is NULL, the **GetProfileString** function enumerates all key names associated with *lpAppName* by filling the location pointed to by *lpReturnedString* with a list of key names (not values). Each key name in the list is terminated with a null character.

Parameters

lpAppName	**LPSTR** Points to a null-terminated character string that names the application.
lpKeyName	**LPSTR** Points to a null-terminated character string that names a key.
lpDefault	**LPSTR** Specifies the default value for the given key if the key cannot be found in the initialization file.
lpReturnedString	**LPSTR** Points to the buffer that receives the character string.
nSize	**int** Specifies the number of characters (including the last null character) that will be copied to the buffer.

Return value The return value specifies the number of characters copied to the buffer identified by the *lpReturnedString* parameter, not including the terminating null character. If the buffer is not large enough to contain the entire string and *lpKeyName* is not NULL, the return value is equal to the length specified by the *nSize* parameter. If the buffer is not large enough to contain the entire string and *lpKeyName* is NULL, the return value is equal to the length specified by the *nSize* parameter minus 2.

Comments **GetProfileString** is not case-dependent, so the strings in *lpAppName* and *lpKeyName* may be in any combination of uppercase and lowercase letters.

GetProp

Syntax HANDLE GetProp(hWnd, lpString)
function GetProp(Wnd: HWnd; Str: PChar): THandle;

This function retrieves a data handle from the property list of the specified window. The character string pointed to by the *lpString* parameter identifies the handle to be retrieved. The string and handle are assumed to have been added to the property list by using the **SetProp** function.

Parameters

hWnd	**HWND** Identifies the window whose property list is to be searched.
lpString	**LPSTR** Points to a null-terminated character string or an atom that identifies a string. If an atom is given, it must have been created previously by using the **AddAtom** function. The atom, a 16-bit value, must be placed in the low-order word of the *lpString* parameter; the high-order word must be set to zero.

Return value The return value identifies the associated data handle if the property list contains the given string. Otherwise, it is NULL.

Comments The value retrieved by the **GetProp** function can be any 16-bit value useful to the application.

GetRgnBox 3.0

Syntax int GetRgnBox(hRgn, lpRect)
function GetRgnBox(Rgn: HRgn; var Rect: TRect): Integer;

This function retrieves the coordinates of the bounding rectangle of the region specified by the *hRgn* parameter.

Parameters

hRgn	**HRGN** Identifies the region.
lpRect	**LPRECT** Points to a **RECT** data structure to receive the coordinates of the bounding rectangle.

Return value The return value specifies the region's type. It can be any of the following values.

Value	Meaning
COMPLEXREGION	Region has overlapping borders.
NULLREGION	Region is empty.
SIMPLEREGION	Region has no overlapping borders.

The return value is NULL if the *hRgn* parameter does not specify a valid region.

GetROP2

Syntax int GetROP2(hDC)
function GetROP2(DC: HDC): Integer;

This function retrieves the current drawing mode. The drawing mode specifies how the pen or interior color and the color already on the display surface are combined to yield a new color.

Parameters *hDC* **HDC** Identifies the device context for a raster device.

Return value The return value specifies the drawing mode. For a list of the drawing modes, see the table "Drawing modes," in the **SetROP2** function, later in this chapter.

Comments For more information about the drawing modes, see Chapter 11, "Binary and ternary raster-operation codes," in *Reference, Volume 2*.

GetRValue

Syntax BYTE GetRValue(rgbColor)
function GetRValue(RGBColor: Longint): Byte;

This macro extracts the red value from an RGB color value.

Parameters *rgbColor* **DWORD** Specifies a red, a green, and a blue color field, each specifying the intensity of the given color.

Return value The return value specifies a byte that contains the red value of the *rgbColor* parameter.

Comments The value 0FFH corresponds to the maximum intensity value for a single byte; 000H corresponds to the minimum intensity value for a single byte.

GetScrollPos

Syntax int GetScrollPos(hWnd, nBar)
function GetScrollPos(Wnd: HWnd; Bar: Integer): Integer;

This function retrieves the current position of a scroll-bar thumb. The current position is a relative value that depends on the current scrolling

range. For example, if the scrolling range is 0 to 100 and the thumb is in the middle of the bar, the current position is 50.

Parameters

hWnd — **HWND** Identifies a window that has standard scroll bars or a scroll-bar control, depending on the value of the *nBar* parameter.

nBar — **int** Specifies the scroll bar to examine. It can be one of the following values:

Value	Meaning
SB_CTL	Retrieves the position of a scroll-bar control. In this case, the *hWnd* parameter must be the window handle of a scroll-bar control.
SB_HORZ	Retrieves the position of a window's horizontal scroll bar.
SB_VERT	Retrieves the position of a window's vertical scroll bar.

Return value The return value specifies the current position of the scroll-bar thumb.

GetScrollRange

Syntax

void GetScrollRange(hWnd, nBar, lpMinPos, lpMaxPos)
procedure GetScrollRange(Wnd: HWnd; Bar: Integer; var MinPos, MaxPos: Integer);

This function copies the current minimum and maximum scroll-bar positions for the given scroll bar to the locations specified by the *lpMinPos* and *lpMaxPos* parameters. If the given window does not have standard scroll bars or is not a scroll-bar control, then the **GetScrollRange** function copies zero to *lpMinPos* and *lpMaxPos*.

Parameters

hWnd — **HWND** Identifies a window that has standard scroll bars or a scroll-bar control, depending on *nBar*'s value.

nBar — **int** Specifies an integer value that identifies which scroll bar to retrieve. It can be one of the following values:

Value	Meaning
SB_CTL	Retrieves the position of a scroll-bar control; in this case, the *hWnd* parameter must be the handle of a scroll-bar control.
SB_HORZ	Retrieves the position of a window's horizontal scroll bar.

	SB_VERT	Retrieves the position of a window's vertical scroll bar.

lpMinPos **LPINT** Points to the integer variable that is to receive the minimum position.

lpMaxPos **LPINT** Points to the integer variable that is to receive the maximum position.

Return value None.

Comments The default range for a standard scroll bar is 0 to 100. The default range for a scroll-bar control is empty (both values are zero).

G

GetStockObject

Syntax HANDLE GetStockObject(nIndex)
function GetStockObject(Index: Integer): THandle;

This function retrieves a handle to one of the predefined stock pens, brushes, or fonts.

Parameters *nIndex* **int** Specifies the type of stock object desired. It can be any one of the following values:

Value	**Meaning**
BLACK_BRUSH	Black brush
DKGRAY_BRUSH	Dark gray brush
GRAY_BRUSH	Gray brush
HOLLOW_BRUSH	Hollow brush
LTGRAY_BRUSH	Light gray brush
NULL_BRUSH	Null brush
WHITE_BRUSH	White brush
BLACK_PEN	Black pen
NULL_PEN	Null pen
WHITE_PEN	White pen
ANSI_FIXED_FONT	ANSI fixed system font
ANSI_VAR_FONT	ANSI variable system font
DEVICE_DEFAULT_FONT	Device-dependent font
OEM_FIXED_FONT	OEM-dependent fixed font
SYSTEM_FONT	The system font. By default, Windows uses the system font to draw menus, dialog-box controls, and other text. In Windows versions 3.0 and later,

	the system font is proportional width; earlier versions of Windows use a fixed-width system font.
SYSTEM_FIXED_FONT	The fixed-width system font used in earlier versions of Windows. This stock object is available for compatibility purposes.
DEFAULT_PALETTE	Default color palette. This palette consists of the 20 static colors always present in the system palette for matching colors in the logical palettes of background windows.

Return value The return value identifies the desired logical object if the function is successful. Otherwise, it is NULL.

Comments The DKGRAY_BRUSH, GRAY_BRUSH, and LTGRAY_BRUSH objects should not be used as background brushes or for any other purpose in a window whose class does not specify CS_HREDRAW and CS_VREDRAW styles. Using a gray stock brush in such windows can lead to misalignment of brush patterns after a window is moved or sized. Stock-brush origins cannot be adjusted (for more information, see the **SetBrushOrg** function, later in this chapter).

GetStretchBltMode

Syntax int GetStretchBltMode(hDC)
function GetStretchBltMode(DC: HDC): Integer;

This function retrieves the current stretching mode. The stretching mode defines how information is to be added or removed from bitmaps that are stretched or compressed by using the **StretchBlt** function.

Parameters *hDC* **HDC** Identifies the device context.

Return value The return value specifies the current stretching mode. It can be WHITEONBLACK, BLACKONWHITE, or COLORONCOLOR. For more information, see the **SetStretchBltMode** function, later in this chapter.

GetSubMenu

Syntax HMENU GetSubMenu(hMenu, nPos)
function GetSubMenu(Menu: HMenu; Pos: Integer): HMenu;

This function retrieves the menu handle of a pop-up menu.

Parameters

hMenu	**HMENU** Identifies the menu.
nPos	**int** Specifies the position in the given menu of the pop-up menu. Position values start at zero for the first menu item. The pop-up menu's integer ID cannot be used in this function.

Return value The return value identifies the given pop-up menu. It is NULL if no pop-up menu exists at the given position.

GetSysColor

Syntax DWORD GetSysColor(nIndex)
function GetSysColor(Index: Integer): TColorRef;

This function retrieves the current color of the display element specified by the *nIndex* parameter. Display elements are the various parts of a window and the Windows display that appear on the system display screen.

Parameters

nIndex	**int** Specifies the display element whose color is to be retrieved. For a list of the index values, see the **SetSysColor** function, later in this chapter.

Return value The return value specifies an RGB color value that names the color of the given element.

Comments System colors for monochrome displays are usually interpreted as various shades of gray.

GetSysModalWindow

Syntax HWND GetSysModalWindow( )
function GetSysModalWindow: HWnd;

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

Parameters None.

Return value The return value identifies the system-modal window, if one is present. If no such window is present, the return value is NULL.

GetSystemDirectory 3.0

Syntax WORD GetSystemDirectory(lpBuffer, nSize)
procedure GetSystemDirectory(Buffer: PChar; Size: Word);

This function obtains the pathname of the Windows system subdirectory. The system subdirectory contains such files as Windows libraries, drivers, and font files.

Parameters

lpBuffer **LPSTR** Points to the buffer that is to receive the null-terminated character string containing the pathname.

nSize **int** Specifies the maximum size (in bytes) of the buffer. This value should be set to at least 144 to allow sufficient room in the buffer for the pathname.

Return value The return value is the length of the string copied to *lpBuffer*, not including the terminating null character. If the return value is greater than *nSize*, the return value is the size of the buffer required to hold the pathname. The return value is zero if the function failed.

Comments The pathname retrieved by this function does not end with a backslash (\ ), unless the system directory is the root directory. For example, if the system directory is named WINDOWS\SYSTEM on drive C:, the pathname of the system subdirectory retrieved by this function is C:\WINDOWS\SYSTEM.

GetSystemMenu

Syntax HMENU GetSystemMenu(hWnd, bRevert)
function GetSystemMenu(Wnd: HWnd; Revert: Bool): HMenu;

This function allows the application to access the System menu for copying and modification.

Parameters

hWnd **HWND** Identifies the window that will own a copy of the System menu.

bRevert **BOOL** Specifies the action to be taken.

If bRevert is:	Description
zero	**GetSystemMenu** returns a handle to a copy of the System menu currently in use. This copy is initially identical to the System menu, but can be modified.
nonzero	**GetSystemMenu** destroys the possibly modified copy of the System menu (if there is one) that belongs to the specified window and returns a handle to the original, unmodified version of the System menu.

Return value The return value identifies the System menu if *bRevert* is nonzero and the System menu has been modified. If *bRevert* is nonzero and the System menu has *not* been modified, the return value is NULL. If *bRevert* is zero, the return value identifies a copy of the System menu.

Comments Any window that does not use the **GetSystemMenu** function to make its own copy of the System menu receives the standard System menu.

The handle returned by the **GetSystemMenu** function can be used with the **AppendMenu, InsertMenu** or **ModifyMenu** functions to change the System menu. The System menu initially contains items identified with various ID values such as SC_CLOSE, SC_MOVE, and SC_SIZE. Menu items on the System menu send WM_SYSCOMMAND messages. All predefined System-menu items have ID numbers greater than 0xF000. If an application adds commands to the System menu, it should use ID numbers less than F000.

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

GetSystemMetrics

Syntax int GetSystemMetrics(nIndex)
function GetSystemMetrics(Index: Integer): Integer;

This function retrieves the system metrics. The system metrics are the widths and heights of various display elements of the Windows display. The **GetSystemMetrics** function can also return flags that indicate

whether the current version is a debugging version, whether a mouse is present, or whether the meaning of the left and right mouse buttons have been exchanged.

Parameters *nIndex* **int** Specifies the system measurement to be retrieved. All measurements are given in pixels. The system measurement must be one of the values listed in Table 4.10, "System Metric Indexes."

Return value The return value specifies the requested system metric.

Comments System metrics depend on the system display and may vary from display to display. Table 4.10 lists the system-metric values for the *nIndex* parameter:

Table 4.10 System metric indexes

Index	Meaning
SM_CXSCREEN	Width of screen.
SM_CYSCREEN	Height of screen.
SM_CXFRAME	Width of window frame that can be sized.
SM_CYFRAME	Height of window frame that can be sized.
SM_CXVSCROLL	Width of arrow bitmap on vertical scroll bar.
SM_CYVSCROLL	Height of arrow bitmap on vertical scroll bar.
SM_CXHSCROLL	Width of arrow bitmap on horizontal scroll bar.
SM_CYHSCROLL	Height of arrow bitmap on horizontal scroll bar.
SM_CYCAPTION	Height of caption.
SM_CXBORDER	Width of window frame that cannot be sized.
SM_CYBORDER	Height of window frame that cannot be sized.
SM_CXDLGFRAME	Width of frame when window has WS_DLGFRAME style.
SM_CYDLGFRAME	Height of frame when window has WS_DLGFRAME style.
SM_CXHTHUMB	Width of thumb box on horizontal scroll bar.
SM_CYVTHUMB	Height of thumb box on vertical scroll bar.
SM_CXICON	Width of icon.
SM_CYICON	Height of icon.
SM_CXCURSOR	Width of cursor.
SM_CYCURSOR	Height of cursor.
SM_CYMENU	Height of single-line menu bar.
SM_CXFULLSCREEN	Width of window client area for full-screen window.
SM_CYFULLSCREEN	Height of window client area for full-screen window (equivalent to the height of the screen minus the height of the window caption).
SM_CYKANJIWINDOW	Height of Kanji window.
SM_CXMINTRACK	Minimum tracking width of window.
SM_CYMINTRACK	Minimum tracking height of window.
SM_CXMIN	Minimum width of window.
SM_CYMIN	Minimum height of window.
SM_CXSIZE	Width of bitmaps contained in the title bar.
SM_CYSIZE	Height of bitmaps contained in the title bar.
SM_MOUSEPRESENT	Nonzero if mouse hardware installed.

Table 4.10: System metric indexes (continued)

SM_DEBUG	Nonzero if Windows debugging version.
SM_SWAPBUTTON	Nonzero if left and right mouse buttons swapped.

GetSystemPaletteEntries 3.0

Syntax WORD GetSystemPaletteEntries(hDC, wStartIndex, wNumEntries, lpPaletteEntries)
function GetSystemPaletteEntries(DC: HDC; StartIndex, NumEntries: Word; var PaletteEntries: TPaletteEntry): Word;

This function retrieves a range of palette entries from the system palette.

Parameters

hDC **HDC** Identifies the device context.

wStartIndex **WORD** Specifies the first entry in the system palette to be retrieved.

wNumEntries **WORD** Specifies the number of entries in the system palette to be retrieved.

lpPaletteEntries **LPPALETTEENTRY** Points to an array of **PALETTEENTRY** data structures to receive the palette entries. The array must contain at least as many data structures as specified by the *wNumEntries* parameter.

Return value The return value is the number of entries retrieved from the system palette. It is zero if the function failed.

GetSystemPaletteUse 3.0

Syntax WORD GetSystemPaletteUse(hDC)
function GetSystemPaletteUse(DC: HDC): Word;

This function determines whether an application has access to the full system palette. By default, the system palette contains 20 static colors which are not changed when an application realizes its logical palette. An application can gain access to most of these colors by calling the **SetSystemPaletteUse** function.

The device context identified by the *hDC* parameter must refer to a device that supports color palettes.

Parameters

hDC **HDC** Identifies the device context.

Return value The return value specifies the current use of the system palette. It is either of the following values:

Value	Meaning
SYSPAL_NOSTATIC	System palette contains no static colors except black and white.
SYSPAL_STATIC	System palette contains static colors which will not change when an application realizes its logical palette.

GetTabbedTextExtent 3.0

Syntax DWORD GetTabbedTextExtent(hDC, lpString, nCount, nTabPositions, lpnTabStopPositions)
function GetTabbedTextExtent(DC: HDC; Str: PChar; Count, TabPositions: Integer; var TabStopPositions): Longint;

This function computes the width and height of the line of text pointed to by the *lpString* parameter. If the string contains one or more tab characters, the width of the string is based upon the tab stops specified by the *lpnTabStopPositions* parameter. The **GetTabbedTextExtent** function uses the currently selected font to compute the dimensions of the string. The width and height (in logical units) are computed without considering the current clipping region.

Parameters

hDC	**HDC** Identifies the device context.
lpString	**LPSTR** Points to a text string.
nCount	**int** Specifies the number of characters in the text string.
nTabPositions	**int** Specifies the number of tab-stop positions in the array to which the *lpnTabStopPositions* points.
lpnTabStopPositions	**LPINT** Points to an array of integers containing the tab-stop positions in pixels. The tab stops must be sorted in increasing order; back tabs are not allowed.

Return value The return value specifies the dimensions of the string. The height is in the high-order word; the width is in the low-order word.

Comments Since some devices do not place characters in regular cell arrays (that is, they carry out kerning), the sum of the extents of the characters in a string may not be equal to the extent of the string.

If the *nTabPositions* parameter is zero and the *lpnTabStopPositions* parameter is NULL, tabs are expanded to eight average character widths.

If *nTabPositions* is 1, the tab stops will be separated by the distance specified by the first value in the array to which *lpnTabStopPositions* points.

If *lpnTabStopPositions* points to more than a single value, then a tab stop is set for each value in the array, up to the number specified by *nTabPositions*.

GetTempDrive

Syntax BYTE GetTempDrive(cDriveLetter)
function GetTempDrive(DriveLetter: Char): Char;

This function takes a drive letter or zero and returns a letter that specifies the optimal drive for a temporary file (the disk drive that can provide the best access time during disk operations with a temporary file).

The **GetTempDrive** function returns the drive letter of a hard disk if the system has one. If the *cDriveLetter* parameter is zero, the function returns the drive letter of the current disk; if *cDriveLetter* is a letter, the function returns the letter of that drive or the letter of another available drive.

Parameters *cDriveLetter* **BYTE** Specifies a disk-drive letter.

Return value The return value specifies the optimal disk drive for temporary files.

GetTempFileName

Syntax int GetTempFileName(cDriveLetter, lpPrefixString, wUnique, lpTempFileName)
function GetTempFileName(DriveLetter: Char; PrefixString: PChar; Unique: Word; TempFileName: PChar): Integer;

This function creates a temporary filename of the following form:*drive*:\ *path**prefixuuuu.tmp*

In this syntax line, *drive* is the drive letter specified by the *cDriveLetter* parameter; *path* is the pathname of the temporary file (either the root directory of the specified drive or the directory specified in the TEMP environment variable); *prefix* is all the letters (up to the first three) of the string pointed to by the *lpPrefixString* parameter; and *uuuu* is the hexadecimal value of the number specified by the *wUnique* parameter.

Parameters

cDriveLetter	**BYTE** Specifies the suggested drive for the temporary filename. If *cDriveLetter* is zero, the default drive is used.
lpPrefixString	**LPSTR** Points to a null-terminated character string to be used as the temporary filename prefix. This string must consist of characters in the OEM-defined character set.
wUnique	**WORD** Specifies an unsigned short integer.
lpTempFileName	**LPSTR** Points to the buffer that is to receive the temporary filename. This string consists of characters in the OEM-defined character set. This buffer should be at least 144 bytes in length to allow sufficient room for the pathname.

Return value The return value specifies a unique numeric value used in the temporary filename. If a nonzero value was given for the *wUnique* parameter, the return value specifies this same number.

Comments To avoid problems resulting from converting OEM character an string to an ANSI string, an application should call the **_lopen** function to create the temporary file.
The **GetTempFileName** function uses the suggested drive letter for creating the temporary filename, except in the following cases:

- If a hard disk is present, **GetTempFileName** always uses the drive letter of the first hard disk.
- Otherwise, if a TEMP environment variable is defined and its value begins with a drive letter, that drive letter is used.

If the TF_FORCEDRIVE bit of the *cDriveLetter* parameter is set, the above exceptions do not apply. The temporary filename will always be created in the current directory of the drive specified by *cDriveLetter*, regardless of the presence of a hard disk or the TEMP environment variable.

If the *wUnique* parameter is zero, **GetTempFileName** attempts to form a unique number based on the current system time. If a file with the resulting filename exists, the number is increased by one and the test for existence is repeated. This continues until a unique filename is found; **GetTempFileName** then creates a file by that name and closes it. No attempt is made to create and open the file when *wUnique* is nonzero.

GetTextAlign

Syntax WORD GetTextAlign(hDC)

function GetTextAlign(DC: HDC): Word;

This function retrieves the status of the text-alignment flags. The text-alignment flags determine how the **TextOut** and **ExtTextOut** functions align a string of text in relation to the string's starting point.

Parameters *hDC* **HDC** Identifies the device context.

Return value The return value specifies the status of the text-alignment flags. The return value is a combination of one or more of the following values:

Value	Meaning
TA_BASELINE	Specifies alignment of the *x*-axis and the baseline of the chosen font within the bounding rectangle.
TA_BOTTOM	Specifies alignment of the *x*-axis and the bottom of the bounding rectangle.
TA_CENTER	Specifies alignment of the *y*-axis and the center of the bounding rectangle.
TA_LEFT	Specifies alignment of the *y*-axis and the left side of the bounding rectangle.
TA_NOUPDATECP	Specifies that the current position is not updated.
TA_RIGHT	Specifies alignment of the *y*-axis and the right side of the bounding rectangle.
TA_TOP	Specifies alignment of the *x*-axis and the top of the bounding rectangle.
TA_UPDATECP	Specifies that the current position is updated.

Comments The text-alignment flags are not necessarily single-bit flags and may be equal to zero. To verify that a particular flag is set in the return value of this function, build an application that will perform the following steps:

1. Apply the bitwise OR operator to the flag and its related flags.

   The following list shows the groups of related flags:

   - TA_LEFT, TA_CENTER, and TA_RIGHT
   - TA_BASELINE, TA_BOTTOM, and TA_TOP
   - TA_NOUPDATECP and TA_UPDATECP
2. Apply the bitwise AND operator to the result and the return value.
3. Test for the equality of this result and the flag.

The following example shows a method for determining which horizontal-alignment flag is set:

```
switch ((TA_LEFT | TA_RIGHT | TA_CENTER) & GetTextAlign(hDC)) { case TA_LEFT
:
   case TA_RIGHT
```

```
:
   case TA_CENTER
:
   }
```

GetTextCharacterExtra

Syntax int GetTextCharacterExtra(hDC)
function GetTextCharacterExtra(DC: HDC): Integer;

This function retrieves the current intercharacter spacing. The intercharacter spacing defines the extra space (in logical units) that the **TextOut** or **ExtTextOut** functions add to each character as they write a line. The spacing is used to expand lines of text.

If the current mapping mode is not MM_TEXT, the **GetTextCharacterExtra** function transforms and rounds the result to the nearest unit.

Parameters *hDC* **HDC** Identifies the device context.

Return value The return value specifies the current intercharacter spacing.

GetTextColor

Syntax DWORD GetTextColor(hDC)
function GetTextColor(DC: HDC): TColorRef;

This function retrieves the current text color. The text color defines the foreground color of characters drawn by using the **TextOut** or **ExtTextOut** functions.

Parameters *hDC* **HDC** Identifies the device context.

Return value The return value specifies the current text color as an RGB color value.

GetTextExtent

Syntax DWORD GetTextExtent(hDC, lpString, nCount)
function GetTextExtent(DC: HDC; Str: PChar; Count: Integer): Longint;

This function computes the width and height of the line of text pointed to by the *lpString* parameter. The **GetTextExtent** function uses the currently selected font to compute the dimensions of the string. The width and

height (in logical units) are computed without considering the current clipping region.

Parameters

hDC	**HDC** Identifies the device context.
lpString	**LPSTR** Points to a text string.
nCount	**int** Specifies the number of characters in the text string.

Return value The return value specifies the dimensions of the string. The height is in the high-order word; the width is in the low-order word.

Comments Since some devices do not place characters in regular cell arrays (that is, they carry out kerning), the sum of the extents of the characters in a string may not be equal to the extent of the string.

GetTextFace

Syntax int GetTextFace(hDC, nCount, lpFacename)
function GetTextFace(DC: HDC; Count: Integer; Facename: PChar): Integer;

This function copies the typeface name of the selected font into a buffer pointed to by the *lpFacename* parameter. The typeface name is copied as a null-terminated character string. The *nCount* parameter specifies the maximum number of characters to be copied. If the name is longer than the number of characters specified by *nCount*, it is truncated.

Parameters

hDC	**HDC** Identifies the device context.
nCount	**int** Specifies the size of the buffer in bytes.
lpFacename	**LPSTR** Points to the buffer that is to receive the typeface name.

Return value The return value specifies the actual number of bytes copied to the buffer. It is zero if an error occurs.

GetTextMetrics

Syntax BOOL GetTextMetrics(hDC, lpMetrics)
function GetTextMetrics(DC: HDC; var Metrics: TTextMetric): Bool;

This function fills the buffer pointed to by the *lpMetrics* parameter with the metrics for the selected font.

Parameters

hDC	**HDC** Identifies the device context.

lpMetrics **LPTEXTMETRIC** Points to the **TEXTMETRIC** data structure that is to receive the metrics.

Return value The return value specifies the outcome of the function. It is nonzero if the function is successful. Otherwise, it is zero.

GetThresholdEvent

Syntax LPINT GetThresholdEvent( )
function GetThresholdEvent: PInteger;

This function retrieves a flag that identifies a recent threshold event. A threshold event is any transition of a voice queue from n to $n - 1$ where n is the threshold level in notes.

Parameters None.

Return value The return value points to a short integer that specifies a threshold event.

GetThresholdStatus

Syntax int GetThresholdStatus( )
function GetThresholdStatus: Integer;

This function retrieves the threshold-event status for each voice. Each bit in the status represents a voice. If a bit is set, the voice-queue level is currently below threshold.

The **GetThresholdStatus** function also clears the threshold-event flag.

Parameters None.

Return value The return value specifies the status flags of the current threshold event.

GetTickCount

Syntax DWORD GetTickCount( )
function GetTickCount: Longint;

This function obtains the number of milliseconds that have elapsed since the system was started.

Parameters None.

Return value The return value specifies the number of milliseconds that have elapsed since the system was started.

Comments The count is accurate within ±55 milliseconds.

GetTopWindow

Syntax HWND GetTopWindow(hWnd)
function GetTopWindow(Wnd: HWnd): HWnd;

This function searches for a handle to the top-level child window that belongs to the parent window associated with the *hWnd* parameter. If the window has no children, this function returns NULL.

Parameters *hWnd* **HWND** Identifies the parent window.

Return value The return value identifies a handle to the top-level child window in a parent window's linked list of child windows. If no child windows exist, it is NULL.

GetUpdateRect

Syntax BOOL GetUpdateRect(hWnd, lpRect, bErase)
function GetUpdateRect(Wnd: HWnd; var Rect: TRect; Erase: Book): Bool;

This function retrieves the coordinates of the smallest rectangle that completely encloses the update region of the given window. If the window was created with the CS_OWNDC style and the mapping mode is not MM_TEXT, the **GetUpdateRect** function gives the rectangle in logical coordinates. Otherwise, **GetUpdateRect** gives the rectangle in client coordinates. If there is no update region, **GetUpdateRect** makes the rectangle empty (sets all coordinates to zero).

The *bErase* parameter specifies whether **GetUpdateRect** should erase the background of the update region. If *bErase* is TRUE and the update region is not empty, the background is erased. To erase the background, **GetUpdateRect** sends a WM_ERASEBKGND message to the given window.

Parameters

hWnd **HWND** Identifies the window whose update region is to be retrieved.

lpRect **LPRECT** Points to the **RECT** data structure that is to receive the client coordinates of the enclosing rectangle.

bErase	**BOOL** Specifies whether the background in the update region is to be erased.

Return value The return value specifies the status of the update region of the given window. It is nonzero if the update region is not empty. Otherwise, it is zero.

Comments The update rectangle retrieved by the **BeginPaint** function is identical to that retrieved by the **GetUpdateRect** function.

BeginPaint automatically validates the update region, so any call to **GetUpdateRect** made immediately after the **BeginPaint** call retrieves an empty update region.

GetUpdateRgn

Syntax int GetUpdateRgn(hWnd, hRgn, fErase)
function GetUpdateRgn(Wnd: HWnd; Rgn: HRgn; Erase: Bool): Integer;

This function copies a window's update region into a region identified by the *hRgn* parameter. The coordinates of this region are relative to the upper-left corner of the window (client coordinates).

Parameters

hWnd	**HWND** Identifies the window that contains the region to be updated.
hRgn	**HRGN** Identifies the update region.
fErase	**BOOL** Specifies whether or not the window background should be erased and nonclient areas of child windows should be drawn. If it is zero, no drawing is done.

Return value The return value specifies a short-integer flag that indicates the type of resulting region. It can be any one of the following values:

Parameters

COMPLEXREGION	The region has overlapping borders.
ERROR	No region was created.
NULLREGION	The region is empty.
SIMPLEREGION	The region has no overlapping borders.

Comments **BeginPaint** automatically validates the update region, so any call to **GetUpdateRgn** made immediately after the **BeginPaint** call retrieves an empty update region.

GetVersion

Syntax WORD GetVersion( )
function GetVersion: Longint;

This function returns the current version number of Windows.

Parameters None.

Return value The return value specifies the major and minor version numbers of Windows. The high-order byte specifies the minor version (revision) number; the low-order byte specifies the major version number.

GetViewportExt

Syntax DWORD GetViewportExt(hDC)
function GetViewportExt(DC: HDC): Longint;

This function retrieves the *x*- and *y*-extents of the device context's viewport.

Parameters *hDC* **HDC** Identifies the device context.

Return value The return value specifies the *x*- and *y*-extents (in device units). The *y*-extent is in the high-order word; the *x*-extent is in the low-order word.

GetViewportOrg

Syntax DWORD GetViewportOrg(hDC)
function GetViewportOrg(DC: HDC): Longint;

This function retrieves the *x*- and *y*-coordinates of the origin of the viewport associated with the specified device context.

Parameters *hDC* **HDC** Identifies the device context.

Return value The return value specifies the origin of the viewport (in device coordinates). The *y*-coordinate is in the high-order word; the *x*-coordinate is in the low-order word.

GetWindow

Syntax HWND GetWindow(hWnd, wCmd)
function GetWindow(Wnd: HWnd; Cmd: Word): HWnd;

This function searches for a handle to a window from the window manager's list. The window-manager's list contains entries for all top-level windows, their associated child windows, and the child windows of any child windows. The *wCmd* parameter specifies the relationship between the window identified by the *hWnd* parameter and the window whose handle is returned.

Parameters

hWnd **HWND** Identifies the original window.

wCmd **WORD** Specifies the relationship between the original window and the returned window. It may be one of the following values:

Value	Meaning
GW_CHILD	Identifies the window's first child window.
GW_HWNDFIRST	Returns the first sibling window for a child window. Otherwise, it returns the first top-level window in the list.
GW_HWNDLAST	Returns the last sibling window for a child window. Otherwise, it returns the last top-level window in the list.
GW_HWNDNEXT	Returns the window that follows the given window on the window manager's list.
GW_HWNDPREV	Returns the previous window on the window manager's list.
GW_OWNER	Identifies the window's owner.

Return value The return value identifies a window. It is NULL if it reaches the end of the window manager's list or if the *wCmd* parameter is invalid.

GetWindowDC

Syntax HDC GetWindowDC(hWnd)
function GetWindowDC(Wnd: HWnd): HDC;

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

GetWindowDC assigns default attributes to the display context each time it retrieves the context. Previous attributes are lost.

Parameters *hWnd* **HWND** Identifies the window whose display context is to be retrieved.

Return value The return value identifies the display context for the given window if the function is successful. Otherwise, it is NULL.

Comments The **GetWindowDC** function is intended to be used for special painting effects within a window's nonclient area. Painting in nonclient areas of any window is not recommended.

The **GetSystemMetrics** function can be used to retrieve the dimensions of various parts of the nonclient area, such as the caption bar, menu, and scroll bars.

After painting is complete, the **ReleaseDC** function must be called to release the display context. Failure to release a window display context will have serious effects on painting requested by applications.

GetWindowExt

Syntax DWORD GetWindowExt(hDC)
function GetWindowExt(DC: HDC): Longint;

This function retrieves the *x*- and *y*-extents of the window associated with the specified device context.

Parameters *hDC* **HDC** Identifies the device context.

Return value The return value specifies the *x*- and *y*-extents (in logical units). The *y*-extent is in the high-order word; the *x*-extent is in the low-order word.

GetWindowLong

Syntax LONG GetWindowLong(hWnd, nIndex)
function GetWindowLong(Wnd: HWnd; Index: Integer): Longint;

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

Parameters *hWnd* **HWND** Identifies the window.

nIndex **int** Specifies the byte offset of the value to be retrieved. It can also be one of the following values:

Value	Meaning
GWL_EXSTYLE	Extended window style.
GWL_STYLE	Window style
GWL_WNDPROC	Long pointer to the window function

Return value The return value specifies information about the given window.

Comments To access any extra four-byte values allocated when the window-class structure was created, use a positive byte offset as the index specified by the *nIndex* parameter, starting at zero for the first four-byte value in the extra space, 4 for the next four-byte value and so on.

GetWindowOrg

Syntax DWORD GetWindowOrg(hDC)
function GetWindowOrg(DC: HDC): Longint;

This function retrieves the *x*- and *y*-coordinates of the origin of the window associated with the specified device context.

Parameters *hDC* **HDC** Identifies the device context.

Return value The return value specifies the origin of the window (in logical coordinates). The *y*-coordinate is in the high-order word; the *x*-coordinate is in the low-order word.

GetWindowRect

Syntax void GetWindowRect(hWnd, lpRect)
procedure GetWindowRect(Wnd: HWnd; var Rect: TRect);

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

Parameters *hWnd* **HWND** Identifies the window.

lpRect **LPRECT** Points to a **RECT** data structure that contains the screen coordinates of the upper-left and lower-right corners of the window.

Return value None.

GetWindowsDirectory 3.0

Syntax WORD GetWindowsDirectory(lpBuffer, nSize)
function GetWindowsDirectory (Buffer: PChar; Size: Word): Word;

This function obtains the pathname of the Windows directory. The Windows directory contains such files as Windows applications, initialization files, and help files.

Parameters *lpBuffer* **LPSTR** Points to the buffer that is to receive the null-terminated character string containing the pathname.

nSize **int** Specifies the maximum size (in bytes) of the buffer. This value should be set to at least 144 to allow sufficient room in the buffer for the pathname.

Return value The return value is the length of the string copied to *lpBuffer*, not including the terminating null character. If the return value is greater than *nSize*, the return value is the size of the buffer required to hold the pathname. The return value is zero if the function failed.

Comments The pathname retrieved by this function does not end with a backslash (\ ), unless the Windows directory is the root directory. For example, if the Windows directory is named WINDOWS on drive C:, the pathname of the Windows directory retrieved by this function is C:\WINDOWS. If Windows was installed in the root directory of drive C:, the pathname retrieved by this function is C:\.

GetWindowTask

Syntax HANDLE GetWindowTask(hWnd)
function GetWindowTask(Wnd: HWnd): THandle;

This function searches for the handle of a task associated with the *hWnd* parameter. A task is any program that executes as an independent unit.

All applications are executed as tasks. Each instance of an application is a task.

Parameters *hWnd* **HWND** Identifies the window for which a task handle is retrieved.

Return value The return value identifies the task associated with a particular window.

GetWindowText

Syntax int GetWindowText(hWnd, lpString, nMaxCount)
function GetWindowText(Wnd: HWnd; Str: PChar; MaxCount: Integer): Integer;

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

Parameters

hWnd	**HWND** Identifies the window or control whose caption or text is to be copied.
lpString	**LPSTR** Points to the buffer that is to receive the copied string.
nMaxCount	**int** Specifies the maximum number of characters to be copied to the buffer. If the string is longer than the number of characters specified in the *nMaxCount* parameter, it is truncated.

Return value The return value specifies the length of the copied string. It is zero if the window has no caption or if the caption is empty.

Comments This function causes a WM_GETTEXT message to be sent to the given window or control.

GetWindowTextLength

Syntax int GetWindowTextLength(hWnd)
function GetWindowTextLength(Wnd: HWnd): Integer;

This function returns the length of the given window's caption title. If the *hWnd* parameter identifies a control, the **GetWindowTextLength** function returns the length of the text within the control instead of the caption.

Parameters *hWnd* **HWND** Identifies the window or control.

Return value The return value specifies the text length. It is zero if no such text exists.

GetWindowWord

Syntax WORD GetWindowWord(hWnd, nIndex)
function GetWindowWord(Wnd: HWnd; Index: Integer): Word;

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

Parameters

hWnd **HWND** Identifies the window.

nIndex **int** Specifies the byte offset of the value to be retrieved. It can also be one of the following values:

Value	**Meaning**
GWW_HINSTANCE	Instance handle of the module that owns the window.
GWW_HWNDPARENT	Handle of the parent window, if any. The **SetParent** function changes the parent window of a child window. An application should not call the **SetWindowLong** function to change the parent of a child window.
GWW_ID	Control ID of the child window.

Return value The return value specifies information about the given window.

Comments To access any extra two-byte values allocated when the window-class structure was created, use a positive byte offset as the index specified by the *nIndex* parameter, starting at zero for the first two-byte value in the extra space, 2 for the next two-byte value and so on.

GetWinFlags 3.0

Syntax DWORD GetWinFlags( )
function GetWinFlags: Longint;

This function returns a 32-bit value containing flags which specify the memory configuration under which Windows is running.

Parameters None.

Return value The return value contains flags specifying the current memory configuration. These flags may be any of the following values:

WF_80x87	System contains an Intel math coprocessor.
WF_CPU086	System CPU is an 8086.
WF_CPU186	System CPU is an 80186.
WF_CPU286	System CPU is an 80286.
WF_CPU386	System CPU is an 80386.
WF_CPU486	System CPU is an 80486.
WF_ENHANCED	Windows is running in 386 enhanced mode. The WF_PMODE flag is always set when WF_ENHANCED is set.
WF_LARGEFRAME	Windows is running in EMS large-frame memory configuration.
WF_PMODE	Windows is running in protected mode. This flag is always set when either WF_ENHANCED or WF_STANDARD is set.
WF_SMALLFRAME	Windows is running in EMS small-frame memory configuration.
WF_STANDARD	Windows is running in standard mode. The WF_PMODE flag is always set when WF_STANDARD is set.

If neither WF_ENHANCED nor WF_STANDARD is set, Windows is running in real mode.

GlobalAddAtom

Syntax ATOM GlobalAddAtom(lpString)
function GlobalAddAtom(Str: PChar): TAtom;

This function adds the character string pointed to by the *lpString* parameter to the atom table and creates a new global atom that uniquely identifies the string. A global atom is an atom that is available to all applications. The atom can be used in a subsequent **GlobalGetAtomName** function to retrieve the string from the atom table.

The **GlobalAddAtom** function stores no more than one copy of a given string in the atom table. If the string is already in the table, the function returns the existing atom value and increases the string's reference count by one. The string's reference count is a number that specifies the number of times **GlobalAddAtom** has been called for a particular string.

Parameters	*lpString*	**LPSTR** Points to the character string to be added to the table. The string must be a null-terminated character string.

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

Comments The atom values returned by **GlobalAddAtom** are within the range 0xC000 to 0xFFFF.

GlobalAlloc

Syntax HANDLE GlobalAlloc(wFlags, dwBytes)
function GlobalAlloc(Flags: Word; Bytes: Longint): THandle;

This function allocates the number of bytes of memory specified by the *dwBytes* parameter from the global heap. The memory can be fixed or moveable, depending on the memory type specified by the *wFlags* parameter.

Parameters *wFlags* **WORD** Specifies one or more flags that tell the **GlobalAlloc** function how to allocate the memory. It can be one or more of the following values:

Value	Meaning
GMEM_DDESHARE	Allocates sharable memory. This is used for dynamic data exchange (DDE) only. Note, however, that Windows automatically discards memory allocated with this attribute when the application that allocated the memory terminates.
GMEM_DISCARDABLE	Allocates discardable memory. Can only be used with GMEM_MOVEABLE.
GMEM_FIXED	Allocates fixed memory.
GMEM_MOVEABLE	Allocates moveable memory. Cannot be used with GMEM_FIXED.
GMEM_NOCOMPACT	Does not compact or discard to satisfy the allocation request.

GMEM_NODISCARD	Does not discard to satisfy the allocation request.
GMEM_NOT_BANKED	Allocates non-banked memory. Cannot be used with GMEM_NOTIFY.
GMEM_NOTIFY	Calls the notification routine if the memory object is ever discarded.
GMEM_ZEROINIT	Initializes memory contents to zero.

Choose GMEM_FIXED or GMEM_MOVEABLE, and then combine others as needed by using the bitwise OR operator.

dwBytes **DWORD** Specifies the number of bytes to be allocated.

Return value The return value identifies the allocated global memory if the function is successful. Otherwise, it is NULL.

Comments If this function is successful, it allocates at least the amount requested. The actual amount allocated may be greater, and the application can use the entire amount. To determine the actual amount allocated, call the **GlobalSize** function.

The largest block of memory that an application can allocate is 1 MB in standard mode and 64 MB in 386 enhanced mode.

GlobalCompact

Syntax DWORD GlobalCompact(dwMinFree)
function GlobalCompact(MinFree: Longint): Longint;

This function generates the number of free bytes of global memory specified by the *dwMinFree* parameter by compacting and, if necessary, discarding from the system's global heap. The function *always* compacts memory before checking for free memory. It then checks the global heap for the number of contiguous free bytes specified by the *dwMinFree* parameter. If the bytes do not exist, the **GlobalCompact** function discards unlocked discardable blocks until the requested space is generated, whenever possible.

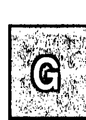
Parameters *dwMinFree* **DWORD** Specifies the number of free bytes desired.

Return value The return value specifies the number of bytes in the largest block of free global memory.

Comments If *dwMinFree* is zero, the return value specifies the number of bytes in the largest free segment that Windows can generate if it removes all discardable segments.

If an application uses the return value as the *dwBytes* parameter to the **GlobalAlloc** function, the GMEM_NOCOMPACT or GMEM_NODISCARD flags should not be used.

GlobalDeleteAtom



Syntax ATOM GlobalDeleteAtom(nAtom)
function GlobalDeleteAtom(AnAtom: TAtom): TAtom;

This function decreases the reference count of a global atom by one. If the atom's reference count becomes zero, this function removes the associated string from the atom table. (A global atom is an atom that is available to all Windows applications.)

An atom's reference count specifies the number of times the atom has been added to the atom table. The **GlobalAddAtom** function increases the count on each call; the **GlobalDeleteAtom** function decreases the count on each call. **GlobalDeleteAtom** removes the string only if the atom's reference count is zero.

Parameters *nAtom* **ATOM** Identifies the atom and character string to be deleted.

Return value The return value specifies the outcome of the function. It is NULL if the function is successful. It is equal to *nAtom* if the function failed and the atom has not been deleted.

GlobalDiscard

Syntax HANDLE GlobalDiscard(hMem)
function GlobalDiscard(Mem: THandle): THandle;

This function discards a global memory block specified by the *hMem* parameter. The lock count of the memory block must be zero.
The global memory block is removed from memory, but its handle remains valid. An application can subsequently pass the handle to the **GlobalReAlloc** function to allocate another global memory block identified by the same handle.

Parameters *hMem* **HANDLE** Identifies the global memory block to be discarded.

Return value The return value identifies the discarded block if the function is successful. Otherwise, it is zero.

Comments The **GlobalDiscard** function discards only global objects that an application allocated with the GMEM_DISCARDABLE and GMEM_MOVEABLE flags set. The function fails if an application attempts to discard a fixed or locked object.

GlobalDosAlloc 3.0

Syntax DWORD GlobalDosAlloc(dwBytes)
function GlobalDosAlloc(Bytes: Longint): Longint;

This function allocates global memory which can be accessed by DOS running in real mode. The memory is guaranteed to exist in the first megabyte of linear address space.

Parameters *dwBytes* **DWORD** Specifies the number of bytes to be allocated.

Return value The return value contains a paragraph-segment value in its high-order word and a selector in its low-order word. An application can use the paragraph-segment value to access memory in real mode and the selector to access memory in protected mode. If Windows is running in real mode, the high-order and low-order words will be equal. If Windows cannot allocate a block of memory of the requested size, the return value is NULL.

Comments An application should not use this function unless it is absolutely necessary. The memory pool from which the object is allocated is a scarce system resource.

GlobalDosFree 3.0

Syntax WORD GlobalDosFree(wSelector)
function GlobalDosFree(Selector: Word): Word;

This function frees a block of global memory previously allocated by a call to the **GlobalDosAlloc** function.

Parameters *wSelector* **WORD** Specifies the memory to be freed.

Return value The return value identifies the outcome of the function. It is NULL if the function is successful. Otherwise, it is equal to *wSelector*.

GlobalFindAtom

Syntax ATOM GlobalFindAtom(lpString)
function GlobalFindAtom(Str: PChar): TAtom;

This function searches the atom table for the character string pointed to by the *lpString* parameter and retrieves the global atom associated with that string. (A global atom is an atom that is available to all Windows applications.)

Parameters *lpString* **LPSTR** Points to the character string to be searched for. The string must be a null-terminated character string.

Return value The return value identifies the global atom associated with the given string. It is NULL if the string is not in the table.

GlobalFix 3.0

Syntax void GlobalFix(hMem)
procedure GlobalFix(Mem: THandle);

This function prevents the global memory block identified by the *hMem* parameter from moving in linear memory. The block is locked into linear memory at its current address and its lock count is increased by one. Locked memory is not subject to moving or discarding except when the memory block is being reallocated by the **GlobalReAlloc** function. The block remains locked in memory until its lock count is decreased to zero.

Each time an application calls **GlobalFix** for a memory object, it must eventually call **GlobalUnfix** for the object. The **GlobalUnfix** function decreases the lock count for the object. Other functions also can affect the lock count of a memory object. See the description of the **GlobalFlags** function for a list of the functions that affect the lock count.

Parameters *hMem* **HANDLE** Identifies the global memory block.

Return value None.

Comments Calling this function interferes with Windows memory management and results in linear-address fragmentation. Very few applications need to fix memory in linear address space.

GlobalFlags

Syntax WORD GlobalFlags(hMem)
function GlobalFlags(Mem: THandle): Word;

This function returns information about the global memory block specified by the *hMem* parameter.

Parameters *hMem* **HANDLE** Identifies the global memory block.

Return value The return value specifies a memory-allocation flag in the high byte. The flag will be one of the following values:

Parameters

GMEM_DDESHARE	The block can be shared. This is used for dynamic data exchange (DDE) only.
GMEM_DISCARDABLE	The block can be discarded.
GMEM_DISCARDED	The block has been discarded.
GMEM_NOT_BANKED	The block cannot be banked.

The low byte of the return value contains the lock count of the block. Use the GMEM_LOCKCOUNT mask to retrieve the lock-count value from the return value.

Comments To test whether or not an object can be discarded, AND the return value of **GlobalFlags** with GMEM_DISCARDABLE.

The following functions can affect the lock count of a global memory block:

Increases Lock Count	Decreases Lock Count
GlobalFix	**GlobalUnfix**
GlobalLock	**GlobalUnlock**
GlobalWire	**GlobalUnWire**
LockSegment	**UnlockSegment**

GlobalFree

Syntax HANDLE GlobalFree(hMem)
function GlobalFree(Mem: THandle): THandle;

This function frees the global memory block identified by the *hMem* parameter and invalidates the handle of the memory block.

Parameters *hMem* **HANDLE** Identifies the global memory block to be freed.

Return value The return value identifies the outcome of the function. It is NULL if the function is successful. Otherwise, it is equal to *hMem*.

Comments The **GlobalFree** function must not be used to free a locked memory block, that is, a memory block with a lock count greater than zero. See the description of the **GlobalFlags** function for a list of the functions that affect the lock count.

GlobalGetAtomName

Syntax WORD GlobalGetAtomName(nAtom, lpBuffer, nSize)
function GlobalGetAtomName(AnAtom: TAtom; Buffer: PChar; Size: Integer): Word;

This function retrieves a copy of the character string associated with the *nAtom* parameter and places it in the buffer pointed to by the *lpBuffer* parameter. The *nSize* parameter specifies the maximum size of the buffer. (A global atom is an atom that is available to all Windows applications.)

Parameters

nAtom	**ATOM** Identifies the character string to be retrieved.
lpBuffer	**LPSTR** Points to the buffer that is to receive the character string.
nSize	**int** Specifies the maximum size (in bytes) of the buffer.

Return value The return value specifies the actual number of bytes copied to the buffer. It is zero if the specified global atom is not valid.

GlobalHandle

Syntax DWORD GlobalHandle(wMem)
function GlobalHandle(Mem: Word): Longint;

This function retrieves the handle of the global memory object whose segment address or selector is specified by the *wMem* parameter.

Parameters

wMem	**WORD** Specifies an unsigned integer value that gives the segment address or selector of a global memory object.

Return value The low-order word of the return value specifies the handle of the global memory object. The high-order word of the return value specifies the segment address or selector of the memory object. The return value is NULL if no handle exists for the memory object.

GlobalLock

Syntax LPSTR GlobalLock(hMem)
function GlobalLock(Mem: THandle): Pointer;

This function retrieves a pointer to the global memory block specified by the *hMem* parameter.
Except for nondiscardable objects in protected (standard or 386 enhanced) mode, the block is locked into memory at the given address and its lock count is increased by one. Locked memory is not subject to moving or discarding except when the memory block is being reallocated by the **GlobalReAlloc** function. The block remains locked in memory until its lock count is decreased to zero.

In protected mode, **GlobalLock** increments the lock count of discardable objects and automatic data segments only.

Each time an application calls **GlobalLock** for an object, it must eventually call **GlobalUnlock** for the object. The **GlobalUnlock** function decreases the lock count for the object if **GlobalLock** increased the lock count for the object. Other functions also can affect the lock count of a memory object. See the description of the **GlobalFlags** function for a list of the functions that affect the lock count.

Parameters *hMem* **HANDLE** Identifies the global memory block to be locked.

Return value The return value points to the first byte of memory in the global block if the function is successful. If the object has been discarded or an error occurs, the return value is NULL.

Comments Discarded objects always have a lock count of zero.

GlobalLRUNewest

Syntax HANDLE GlobalLRUNewest(hMem)
function GlobalLRUNewest(Mem: THandle): THandle;

This function moves the global memory object identified by *hMem* to the newest least-recently-used (LRU) position in memory. This greatly reduces the likelihood that the object will be discarded soon, but does not prevent the object from eventually being discarded.

Parameters *hMem* **HANDLE** Identifies the global memory object to be moved.

Return value The return value is NULL if the *hMem* parameter does not specify a valid handle.

Comments This function is useful only if *hMem* is discardable.

GlobalLRUOldest

Syntax HANDLE GlobalLRUOldest(hMem)
function GlobalLRUOldest(Mem: THandle): THandle;

This routine moves the global memory object identified by *hMem* to the oldest least-recently-used (LRU) position in memory and, in so doing, makes it the next candidate for discarding.

Parameters *hMem* **HANDLE** Identifies the global memory object to be moved.

Return value The return value is NULL if the *hMem* parameter does not specify a valid handle.

Comments This function is useful only if *hMem* is discardable.

GlobalNotify

Syntax void GlobalNotify(lpNotifyProc)
procedure GlobalNotify(NotifyProc: TFarProc);

This function installs a notification procedure for the current task. Windows calls the notification procedure whenever a global memory block allocated with the GMEM_NOTIFY flag is about to be discarded.

Parameters *lpNotifyProc* **FARPROC** Is the procedure instance address of the current task's notification procedure.

Return value None.

Comments An application must not call **GlobalNotify** more than once per instance.

Windows does not call the notification procedure when it discards memory belonging to a DLL.

If the object is discarded, the application must use the GMEM_NOTIFY flag when it recreates the object by calling the **GlobalRealloc** function. Otherwise, the application will not be notified when the object is discarded again.

If the notification procedure returns a nonzero value, Windows discards the global memory block. If it returns zero, the block is not discarded.

The callback function must use the Pascal calling convention and must be declared **FAR**. The callback function must reside in a fixed code segment of a DLL.

Callback function

Bool FAR PASCAL *NotifyProc(hMem)*

NotifyProc is a placeholder for the application-supplied function name. Export the name by including it in an EXPORTS statement in the DLL's module-definition statement.

Parameters *hMem* **HANDLE** Identifies the global memory block being discarded.

Return value The function returns a nonzero value if Windows is to discard the memory block, and zero if it should not.

Comments The callback function is not necessarily called in the context of the application that owns the routine. For this reason, the callback function should not assume the stack segment of the application. The callback function should not call any routine that might move memory.

GlobalPageLock 3.0

Syntax WORD GlobalPageLock(wSelector)
function GlobalPageLock(Selector: THandle): Word;

This function increments the page-lock count of the memory associated with the specified global selector. As long as its page-lock count is nonzero, the data which the selector references is guaranteed to remain in memory at the same physical address and to remain paged in.

GlobalPageLock increments the page-lock count for the block of memory, and the **GlobalPageUnlock** function decrements the page-lock count. Page-locking operations can be nested, but each page lock must be balanced by a corresponding unlock.

Parameters *wSelector* **WORD** Specifies the selector of the memory to be page-locked.

Return value The return value specifies the page-lock count after the function has incremented it. If the function fails, the return value is zero.

Comments An application should not use this function unless it is absolutely necessary. Use of this function violates preferred Windows programming practices. It is intended to be used for dynamically allocated data that

must be accessed at interrupt time. For this reason, it must only be called from a DLL.

GlobalPageUnlock 3.0

Syntax WORD GlobalPageUnlock(wSelector)
function GlobalPageUnlock(Selector: THandle): Word;

This function decrements the page-lock count for the block of memory identified by the *wSelector* parameter and, if the page-lock count reaches zero, allows the block of memory to move and to be paged to disk.

The **GlobalPageLock** function increments the page-lock count for the block of memory, and **GlobalPageUnlock** decrements the page-lock count. Page-locking operations can be nested, but each page lock must be balanced by a corresponding unlock.

Only libraries can call this function.

Parameters *wSelector* **WORD** Specifies the selector of the memory to be page-unlocked.

Return value The return value specifies the page-lock count after the function has decremented it. If the function fails, the return value is zero.

GlobalReAlloc

Syntax HANDLE GlobalReAlloc(hMem, dwBytes, wFlags)
function GlobalReAlloc(Mem: THandle; Bytes: Longint; Flags: Word): THandle;

This function reallocates the global memory block specified by the *hMem* parameter by increasing or decreasing its size to the number of bytes specified by the *dwBytes* parameter.

Parameters

hMem **HANDLE** Identifies the global memory block to be reallocated.

dwBytes **DWORD** Specifies the new size of the memory block.

wFlags **WORD** Specifies how to reallocate the global block.
If the existing memory flags can be modified, use either one or both of the following flags (if both flags are specified, join them with the bitwise OR operator):

Value	Meaning
GMEM_DISCARDABLE	Memory can be discarded. Use only with GMEM_MODIFY.
GMEM_MODIFY	Memory flags are modified. The *dwBytes* parameter is ignored. Use only if an application will modify existing memory flags and not reallocate the memory block to a new size.
GMEM_MOVEABLE	Memory is movable. If *dwBytes* is zero, this flag causes an object previously allocated as moveable and discardable to be discarded if the block's lock count is zero. If the block is not moveable and discardable, the **GlobalReAlloc** will fail. If *dwBytes* is nonzero and the block specified by *hMem* is fixed, this flag allows the reallocated block to be moved to a new fixed location. If a moveable object is locked, this flag allows the object to be moved. This may occur even if the object is currently locked by a previous call to **GlobalLock**. (Note that the handle returned by the **GlobalReAlloc** function in this case may be different from the handle passed to the function.) Use this flag with GMEM_MODIFY to make a fixed memory block moveable.
GMEM_NOCOMPACT	Memory will not be compacted or discarded in order to satisfy the allocation request. This flag is ignored if the GMEM_MODIFY flag is set.
GMEM_NODISCARD	Objects will not be discarded in order to satisfy the allocation request. This flag is ignored if the GMEM_MODIFY flag is set.

GMEM_ZEROINIT	If the block is growing, the additional memory contents are initialized to zero. This flag is ignored if the GMEM_MODIFY flag is set.

Return value The return value identifies the reallocated global memory if the function is successful. The return value is NULL if the block cannot be reallocated.

If the function is successful, the return value is always identical to the *hMem* parameter, unless any of the following conditions is true:

- The GMEM_MOVEABLE flag is used to allow movement of a fixed block to a new fixed location.
- Windows is running in standard mode and the object is reallocated past a multiple of 65,519 bytes (16 bytes less than 64K).
- Windows is running in 386 enhanced mode and the object is reallocated past a multiple of 64K.

GlobalSize

Syntax DWORD GlobalSize(hMem)
function GlobalSize(Mem: THandle): Longint;

This function retrieves the current size (in bytes) of the global memory block specified by the *hMem* parameter.

Parameters *hMem* **HANDLE** Identifies the global memory block.

Return value The return value specifies the actual size (in bytes) of the specified memory block. It is zero if the given handle is not valid or if the object has been discarded.

Comments The actual size of a memory block is sometimes larger than the size requested when the memory was allocated.

An application should call the **GlobalFlags** function prior to calling the **GlobalSize** function in order to verify that the specified memory block was not discarded. If the memory block were discarded, the return value for **GlobalSize** would be meaningless.

GlobalUnfix 3.0

Syntax BOOL GlobalUnfix(hMem)

function GlobalUnfix(Mem: THandle): Bool;

This function unlocks the global memory block specified by the *hMem* parameter.

GlobalUnfix decreases the block's lock count by one. The block is completely unlocked and subject to moving or discarding if the lock count is decreased to zero. Other functions also can affect the lock count of a memory object. See the description of the **GlobalFlags** function for a list of the functions that affect the lock count.

Each time an application calls **GlobalFix** for an object, it must eventually call **GlobalUnfix** for the object.

Parameters *hMem* **HANDLE** Identifies the global memory block to be unlocked.

Return value The return value specifies the outcome of the function. It is zero if the block's lock count was decreased to zero. Otherwise, the return value is nonzero.

GlobalUnlock

Syntax BOOL GlobalUnlock(hMem)
function GlobalUnlock(Mem: THandle): Bool;

This function unlocks the global memory block specified by the *hMem* parameter.

In real mode, or if the block is discardable, **GlobalUnlock** decreases the block's lock count by one. In protected mode, **GlobalUnock** decreases the lock count of discardable objects and automatic data segments only.

The block is completely unlocked and subject to moving or discarding if the lock count is decreased to zero. Other functions also can affect the lock count of a memory object. See the description of the **GlobalFlags** function for a list of the functions that affect the lock count.

In all cases, each time an application calls **GlobalLock** for an object, it must eventually call **GlobalUnlock** for the object.

Parameters *hMem* **HANDLE** Identifies the global memory block to be unlocked.

Return value The return value specifies the outcome of the function. It is zero if the block's lock count was decreased to zero. Otherwise, the return value is nonzero. An application should not rely on the return value to determine the number of times it must subsequently call **GlobalUnlock** for the memory block.

GlobalUnWire

Syntax BOOL GlobalUnWire(hMem)
function GlobalUnWire(Mem: THandle): Bool;

This function unlocks a memory segment that was locked by the **GlobalWire** function and decreases the lock count by one.

The block is completely unlocked and subject to moving or discarding if the lock count is decreased to zero. Other functions also can affect the lock count of a memory object. See the description of the **GlobalFlags** function for a list of the functions that affect the lock count.

Each time an application calls **GlobalWire** for an object, it must eventually call **GlobalUnWire** for the object.

Parameters *hMem* **HANDLE** Identifies the segment that will be unlocked.

Return value The return value specifies the outcome of the function. It is TRUE if the memory segment was unlocked, that is, its lock count was decreased to zero. Otherwise, it is FALSE.

GlobalWire

Syntax LPSTR GlobalWire(hMem)
function GlobalWire(Mem: THandle): Pointer;

This function moves a segment into low memory and locks it—a procedure that is extremely useful if an application must lock a segment for a long period of time. If a segment from the middle portion of memory is locked for a long period of time, it causes memory-management problems by reducing the size of the largest, contiguous available block of memory. The **GlobalWire** function moves a segment to the lowest possible address in memory and locks it, thereby freeing the memory area Windows uses most often.

Each time an application calls **GlobalWire** for an object, it must eventually call **GlobalUnWire** for the object. The **GlobalUnWire** function decreases the lock count for the object. Other functions also can affect the lock count of a memory object. See the description of the **GlobalFlags** function for a list of the functions that affect the lock count.

An application must not call the **GlobalUnlock** function to unlock the object.

Parameters *hMem* **HANDLE** Identifies the segment that will be moved and locked.

Return value The return value points to the new segment location. It is NULL if the function failed.

GrayString

Syntax BOOL GrayString(hDC, hBrush, lpOutputFunc, lpData, nCount, X, Y, nWidth, nHeight)
function GrayString(DC: HDC; Brush: HBrush; OutputFunc: TFarProc; Data: Longint; Count, X, Y, Width, Height: Integer): Bool;

This function draws gray text at the given location. The **GrayString** function draws gray text by writing the text in a memory bitmap, graying the bitmap, and then copying the bitmap to the display. The function grays the text regardless of the selected brush and background. **GrayString** uses the font currently selected for the device context specified by the *hDC* parameter.

If the *lpOutputFunc* parameter is NULL, GDI uses the **TextOut** function, and the *lpData* parameter is assumed to be a long pointer to the character string to be output. If the characters to be output cannot be handled by **TextOut** (for example, the string is stored as a bitmap), the application must supply its own output function.

Parameters

hDC **HDC** Identifies the device context.

hBrush **HBRUSH** Identifies the brush to be used for graying.

lpOutputFunc **FARPROC** Is the procedure-instance address of the application-supplied function that will draw the string, or, if the **TextOut** function is to be used to draw the string, it is a NULL pointer. See the following "Comments" section for details.

lpData **DWORD** Specifies a long pointer to data to be passed to the output function. If the *lpOutputFunc* parameter is NULL, *lpData* must be a long pointer to the string to be output.

nCount **int** Specifies the number of characters to be output. If the *nCount* parameter is zero, **GrayString** calculates the length of the string (assuming that *lpData* is a pointer to the string). If *nCount* is –1 and the function pointed to by *lpOutputFunc* returns zero, the image is shown but not grayed.

X	**int** Specifies the logical *x*-coordinate of the starting position of the rectangle that encloses the string.
Y	**int** Specifies the logical *y*-coordinate of the starting position of the rectangle that encloses the string.
nWidth	**int** Specifies the width (in logical units) of the rectangle that encloses the string. If the *nWidth* parameter is zero, **GrayString** calculates the width of the area, assuming *lpData* is a pointer to the string.
nHeight	**int** Specifies the height (in logical units) of the rectangle that encloses the string. If the *nHeight* parameter is zero, **GrayString** calculates the height of the area, assuming *lpData* is a pointer to the string.

Return value The return value specifies the outcome of the function. It is nonzero if the string is drawn. A return value of zero means that either the **TextOut** function or the application-supplied output function returned zero, or there was insufficient memory to create a memory bitmap for graying.

Comments An application can draw grayed strings on devices that support a solid gray color, without calling the **GrayString** function. The system color COLOR_GRAYTEXT is the solid-gray system color used to draw disabled text. The application can call the **GetSysColor** function to retrieve the color value of COLOR_GRAYTEXT. If the color is other than zero (black), the application can call the **SetTextColor** to set the text color to the color value and then draw the string directly. If the retrieved color is black, the application must call **GrayString** to gray the text.

The callback function must use the Pascal calling convention and must be declared **FAR**.

Callback function

BOOL FAR PASCAL *OutputFunc(hDC, lpData, nCount)*
HDC *hDC;*
DWORD *lpData;*
int *nCount;*

OutputFunc is a placeholder for the application-supplied callback function name. The actual name must be exported by including it in an **EXPORTS** statement in the application's module-definition file.

Parameters

hDC	Identifies a memory device context with a bitmap of at least the width and height specified by the *nWidth* and *nHeight* parameters, respectively.

lpData	Points to the character string to be drawn.
nCount	Specifies the number of characters to be output.

Return value The return value must be nonzero to indicate success. Otherwise, it is zero.

Comments This output function (*OutputFunc*) must draw an image relative to the coordinates (0,0) rather than (*X*,*Y*). The address passed as the *lpOutputFunc* parameter must be created by using the **MakeProcInstance** function, and the output function name must be exported; it must be explicitly defined in an **EXPORTS** statement of the application's module-definition file.

The MM_TEXT mapping mode must be selected before using this function.

HIBYTE

Syntax BYTE HIBYTE(nInteger)
function HiByte(A: Word): Byte;

This macro retrieves the high-order byte from the integer value specified by the *nInteger* parameter.

Parameters *nInteger* **int** Specifies the value to be converted.

Return value The return value specifies the high-order byte of the given value.

HideCaret

Syntax void HideCaret(hWnd)
procedure HideCaret(Wnd: HWnd);

This function hides the caret by removing it from the display screen. Although the caret is no longer visible, it can be displayed again by using the **ShowCaret** function. Hiding the caret does not destroy its current shape.

The **HideCaret** function hides the caret only if the given window owns the caret. If the *hWnd* parameter is NULL, the function hides the caret only if a window in the current task owns the caret.

Hiding is cumulative. If **HideCaret** has been called five times in a row, **ShowCaret** must be called five times before the caret will be shown.

Parameters *hWnd* **HWND** Identifies the window that owns the caret, or it is NULL to indirectly specify the window in the current task that owns the caret.

Return value None.

HiliteMenuItem

Syntax BOOL HiliteMenuItem(hWnd, hMenu, wIDHiliteItem, wHilite)
function HiliteMenuItem(Wnd: HWnd; Menu: HMenu; IDHilite, Hilite: Word): Bool;

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

Parameters

hWnd **HWND** Identifies the window that contains the menu.

hMenu **HMENU** Identifies the top-level menu that contains the item to be highlighted.

wIDHiliteItem **WORD** Specifies the integer identifier of the menu item or the offset of the menu item in the menu, depending on the value of the *wHilite* parameter.

wHilite **WORD** Specifies whether the menu item is highlighted or the highlight is removed. It can be a combination of MF_HILITE or MF_UNHILITE with MF_BYCOMMAND or MF_BYPOSITION. The values can be combined using

the bitwise OR operator. These values have the following meanings:

Value	Meaning
MF_BYCOMMAND	Interprets *wIDHiliteItem* as the menu-item ID (the default interpretation).
MF_BYPOSITION	Interprets *wIDHiliteItem* as an offset.
MF_HILITE	Highlights the item. If this value is not given, highlighting is removed from the item.
MF_UNHILITE	Removes highlighting from the item.

Return value The return value specifies whether or not the menu item is highlighted the outcome of the function. It is nonzero if the item is highlightedwas set to the specified highlight state. Otherwise, it is zero FALSE.

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

HIWORD

Syntax WORD HIWORD(dwInteger)
function HiWord(A: Longint): Word;

This macro retrieves the high-order word from the 32-bit integer value specified by the dwInteger parameter.

Parameters *dwInteger* **DWORD** Specifies the value to be converted.

Return value The return value specifies the high-order word of the given 32-bit integer value.

InflateRect

Syntax void InflateRect(lpRect, X, Y)
procedure InflateRect(var Rect: TRect; X, Y: Integer);

This function increases or decreases the width and height of the specified rectangle. The **InflateRect** function adds *X* units to the left and right ends of the rectangle, and adds *Y* units to the top and bottom. The *X* and *Y* parameters are signed values; positive values increase the width and height, and negative values decrease them.

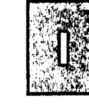
Parameters *lpRect* **LPRECT** Points to the **RECT** data structure to be modified.

X **int** Specifies the amount to increase or decrease the rectangle width. It must be negative to decrease the width.

Y **int** Specifies the amount to increase or decrease the rectangle height. It must be negative to decrease the height.

Return value None.

Comments The coordinate values of a rectangle must not be greater than 32,767 units or less than –32,768 units. The *X* and *Y* parameters must be chosen carefully to prevent invalid rectangles.



InitAtomTable

Syntax BOOL InitAtomTable(nSize)
function InitAtomTable(Size: Integer): Bool;

This function initializes an atom hash table and sets its size to that specified by the *nSize* parameter. If this function is not called, the atom hash table size is set to 37 by default.

If used, this function should be called before any other atom-management function.

Parameters *nSize* **int** Specifies the size (in table entries) of the atom hash table. This value should be a prime number.

Return value The return value specifies the outcome of the function. It is nonzero if the function is successful. Otherwise, it is zero.

Comments If an application uses a large number of atoms, it can reduce the time required to add an atom to the atom table or to find an atom in the table by increasing the size of the table. However, this increases the amount of memory required to maintain the table.

The size of the global atom table cannot be changed from its default size of 37.

InSendMessage

Syntax BOOL InSendMessage( )
function InSendMessage: Bool;

This function specifies whether the current window function is processing a message that is passed to it through a call to the **SendMessage** function.

Parameters None.

Return value The return value specifies the outcome of the function. It is TRUE if the window function is processing a message sent to it with **SendMessage**. Otherwise, it is FALSE.

Comments Applications use the **InSendMessage** function to determine how to handle errors that occur when an inactive window processes messages. For example, if the active window uses **SendMessage** to send a request for information to another window, the other window cannot become active until it returns control from the **SendMessage** call. The only method an inactive window has to inform the user of an error is to create a message box.

InsertMenu 3.0

Syntax BOOL InsertMenu(hMenu, nPosition, wFlags, wIDNewItem, lpNewItem)
function InsertMenu(Menu:HMenu; Position, Flags, IDNewItem: Word; NewItem: PChar): Bool;

This function inserts a new menu item at the position specified by the *nPosition* parameter, moving other items down the menu. The application can specify the state of the menu item by setting values in the *wFlags* parameter.

Parameters

hMenu **HMENU** Identifies the menu to be changed.

nPosition **WORD** Specifies the menu item before which the new menu item is to be inserted. The interpretation of the *nPosition* parameter depends upon the setting of the *wFlags* parameter.

If wFlags is:	nPosition:
MF_BYPOSITION	Specifies the position of the existing menu item. The first item in the menu is at position zero. If *nPosition* is –1, the new menu item is appended to the end of the menu.
MF_BYCOMMAND	Specifies the command ID of the existing menu item.

wFlags **WORD** Specifies how the *nPosition* parameter is interpreted and information about the state of the new menu item when

it is added to the menu. It consists of one or more values listed in the following "Comments" section.

wIDNewItem **WORD** Specifies either the command ID of the new menu item or, if *wFlags* is set to MF_POPUP, the menu handle of the pop-up menu.

lpNewItem **LPSTR** Specifies the content of the new menu item. If *wFlags* is set to MF_STRING (the default), then *lpNewItem* is a long pointer to a null-terminated character string. If *wFlags* is set to MF_BITMAP instead, then *lpNewItem* contains a bitmap handle (**HBITMAP**) in its low-order word. If *wFlags* is set to MF_OWNERDRAW, *lpNewItem* specifies an application-supplied 32-bit value which the application can use to maintain additional data associated with the menu item. This 32-bit value is available to the application in the **itemData** field of the data structure pointed to by the *lParam* parameter of the following messages:

- WM_MEASUREITEM
- WM_DRAWITEM

These messages are sent when the menu item is initially displayed, or is changed.

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

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

Each of the following groups lists flags that should not be used together:

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

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

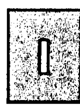
Parameters

MF_BITMAP	Uses a bitmap as the item. The low-order word of the *lpNewItem* parameter contains the handle of the bitmap.
MF_BYCOMMAND	Specifies that the *nPosition* parameter gives the menu-item control ID number (default).

MF_BYPOSITION	Specifies that the *nPosition* parameter gives the position of the menu item to be changed rather than an ID number.
MF_CHECKED	Places a checkmark next to the menu item. If the application has supplied checkmark bitmaps (see the **SetMenuItemBitmaps** function), setting this flag displays the "checkmark on" bitmap next to the menu item.
MF_DISABLED	Disables the menu item so that it cannot be selected, but does not gray it.
MF_ENABLED	Enables the menu item so that it can be selected and restores it from its grayed state.
MF_GRAYED	Disables the menu item so that it cannot be selected and grays it.
MF_MENUBARBREAK	Same as MF_MENUBREAK except that for pop-up menus, separates the new column from the old column with a vertical line.
MF_MENUBREAK	Places the menu item on a new line for static menu-bar items. For pop-up menus, places the menu item in a new column, with no dividing line between the columns.
MF_OWNERDRAW	Specifies that the item is an owner-draw item. The window that owns the menu receives a WM_MEASUREITEM message when the menu is displayed for the first time to retrieve the height and width of the menu item. The WM_DRAWITEM message is then sent to the owner whenever the owner must update the visual appearance of the menu item. This option is not valid for a top-level menu item.
MF_POPUP	Specifies that the menu item has a pop-up menu associated with it. The *wIDNewItem* parameter specifies a handle to a pop-up menu to be associated with the item. Use the MF_OWNERDRAW flag to add either a top-level pop-up menu or a hierarchical pop-up menu to a pop-up menu item.
MF_SEPARATOR	Draws a horizontal dividing line. You can use this flag in a pop-up menu. This line cannot be grayed, disabled, or highlighted. Windows ignores the *lpNewItem* and *wIDNewItem* parameters.

MF_STRING	Specifies that the menu item is a character string; the *lpNewItem* parameter points to the string for the item.
MF_UNCHECKED	Does not place a checkmark next to the item (default). If the application has supplied checkmark bitmaps (see **SetMenuItemBitmaps**), setting this flag displays the "checkmark off" bitmap next to the menu item.

IntersectClipRect



Syntax int IntersectClipRect(hDC, X1, Y1, X2, Y2)
function IntersectClipRect(DC: HDC; X1, Y1, X2, Y2: Integer): Integer;

This function creates a new clipping region by forming the intersection of the current region and the rectangle specified by *X1*, *Y1*, *X2*, and *Y2*. GDI clips all subsequent output to fit within the new boundary.

Parameters

hDC	**HDC** Identifies the device context.
X1	**int** Specifies the logical *x*-coordinate of the upper-left corner of the rectangle.
Y1	**int** Specifies the logical *y*-coordinate of the upper-left corner of the rectangle.
X2	**int** Specifies the logical *x*-coordinate of the lower-right corner of the rectangle.
Y2	**int** Specifies the logical *y*-coordinate of the lower-right corner of the rectangle.

Return value The return value specifies the new clipping region's type. It can be any one of the following values:

Value	Meaning
COMPLEXREGION	New clipping region has overlapping borders.
ERROR	Device context is not valid.
NULLREGION	New clipping region is empty.
SIMPLEREGION	New clipping region has no overlapping borders.

Comments The width of the rectangle, specified by the absolute value of *X2* – *X1*, must not exceed 32,767 units. This limit applies to the height of the rectangle as well.

IntersectRect

Syntax int IntersectRect(lpDestRect, lpSrc1Rect, lpSrc2Rect)
function IntersectRect(var DestRect, Src1Rect, Src2Rect: TRect): Integer;

This function creates the intersection of two existing rectangles. The intersection is the largest rectangle contained in both rectangles. The **IntersectRect** function copies the new rectangle to the **RECT** data structure pointed to by the *lpDestRect* parameter.

Parameters

lpDestRect	**LPRECT** Points to the **RECT** data structure that is to receive the intersection.
lpSrc1Rect	**LPRECT** Points to a **RECT** data structure that contains a source rectangle.
lpSrc2Rect	**LPRECT** Points to a **RECT** data structure that contains a source rectangle.

Return value The return value specifies the intersection of two rectangles. It is nonzero if the intersection of the two rectangles is not empty. It is zero if the intersection is empty.

InvalidateRect

Syntax void InvalidateRect(hWnd, lpRect, bErase)
procedure InvalidateRect(Wnd: HWnd; Rect: PRect; Erase: Bool);

This function invalidates the client area within the given rectangle by adding that rectangle to the window's update region. The invalidated rectangle, along with all other areas in the update region, is marked for painting when the next WM_PAINT message occurs. The invalidated areas accumulate in the update region until the region is processed when the next WM_PAINT message occurs, or the region is validated by using the **ValidateRect** or **ValidateRgn** function.

The *bErase* parameter specifies whether the background within the update area is to be erased when the update region is processed. If *bErase* is nonzero, the background is erased when the **BeginPaint** function is called; if *bErase* is zero, the background remains unchanged. If *bErase* is nonzero for any part of the update region, the background in the entire region is erased, not just in the given part.

Parameters

hWnd	**HWND** Identifies the window whose update region is to be modified.

lpRect **LPRECT** Points to a **RECT** data structure that contains the rectangle (in client coordinates) to be added to the update region. If the *lpRect* parameter is NULL, the entire client area is added to the region.

bErase **BOOL** Specifies whether the background within the update region is to be erased.

Return value None.

Comments Windows sends a WM_PAINT message to a window whenever its update region is not empty and there are no other messages in the application queue for that window.

InvalidateRgn

Syntax void InvalidateRgn(hWnd, hRgn, bErase)
procedure InvalidateRgn(Wnd: HWnd; Rgn: HRgn; Erase: Bool);

This function invalidates the client area within the given region by adding it to the current update region of the given window. The invalidated region, along with all other areas in the update region, is marked for painting when the next WM_PAINT message occurs. The invalidated areas accumulate in the update region until the region is processed when the next WM_PAINT message occurs, or the region is validated by using the **ValidateRect** or **ValidateRgn** function.

The *bErase* parameter specifies whether the background within the update area is to be erased when the update region is processed. If *bErase* is nonzero, the background is erased when the **BeginPaint** function is called; if *bErase* is zero, the background remains unchanged. If *bErase* is nonzero for any part of the update region, the background in the entire region is erased, not just in the given part.

Parameters

hWnd **HWND** Identifies the window whose update region is to be modified.

hRgn **HRGN** Identifies the region to be added to the update region. The region is assumed to have client coordinates.

bErase **BOOL** Specifies whether the background within the update region is to be erased.

Return value None.

Comments Windows sends a WM_PAINT message to a window whenever its update region is not empty and there are no other messages in the application queue for that window.

The given region must have been previously created by using one of the region functions (for more information, see Chapter 1, "Window manager interface functions").

InvertRect

Syntax void InvertRect(hDC, lpRect)
procedure InvertRect(DC: HDC; var Rect: TRect);

This function inverts the contents of the given rectangle. On monochrome displays, the **InvertRect** function makes white pixels black, and black pixels white. On color displays, the inversion depends on how colors are generated for the display. Calling **InvertRect** twice with the same rectangle restores the display to its previous colors.

Parameters

hDC	**HDC** Identifies the device context.
lpRect	**LPRECT** Points to a **RECT** data structure that contains the logical coordinates of the rectangle to be inverted.

Return value None.

Comments The **InvertRect** function compares the values of the **top**, **bottom**, **left**, and **right** fields of the specified rectangle. If **bottom** is less than or equal to **top**, or if **right** is less than or equal to **left**, the rectangle is not drawn.

InvertRgn

Syntax BOOL InvertRgn(hDC, hRgn)
function InvertRgn(DC: HDC; Rgn: HRgn): Bool;

This function inverts the colors in the region specified by the *hRgn* parameter. On monochrome displays, the **InvertRgn** function makes white pixels black, and black pixels white. On color displays, the inversion depends on how the colors are generated for the display.

Parameters

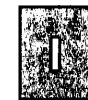
hDC	**HDC** Identifies the device context for the region.
hRgn	**HRGN** Identifies the region to be filled. The coordinates for the region are specified in device units.

Return value The return value specifies the outcome of the function. It is nonzero if the function is successful. Otherwise, it is zero.

IsCharAlpha 3.0

Syntax BOOL IsCharAlpha(cChar)
function IsCharAlpha(Chr: Char): Bool;

This function determines whether a character is an alphabetical character. This determination is made by the language driver based on the criteria of the current language selected by the user at setup or with the Control Panel.



Parameters *cChar* **char** Specifies the character to be tested.

Return value The return value is TRUE if the character is alphabetical. Otherwise, it is FALSE.

IsCharAlphaNumeric 3.0

Syntax BOOL IsCharAlphaNumeric(cChar)
function IsCharAlphaNumeric(Chr: Char): Bool;

This function determines whether a character is an alphabetical or numerical character. This determination is made by the language driver based on the criteria of the current language selected by the user at setup or with the Control Panel.

Parameters *cChar* **char** Specifies the character to be tested.

Return value The return value is TRUE if the character is an alphanumeric character. Otherwise, it is FALSE.

IsCharLower 3.0

Syntax BOOL IsCharLower(cChar)
function IsCharLower(Chr: Char): Bool;

This function determines whether a character is a lowercase character. This determination is made by the language driver based on the criteria of the current language selected by the user at setup or with the Control Panel.

Parameters *cChar* **char** Specifies the character to be tested.

Return value The return value is TRUE if the character is lowercase. Otherwise, it is FALSE.

IsCharUpper 3.0

Syntax BOOL IsCharUpper(cChar)
function IsCharUpper(Chr: Char): Bool;

This function determines whether a character is an uppercase character. This determination is made by the language driver based on the criteria of the current language selected by the user at setup or with the Control Panel.

Parameters *cChar* **char** Specifies the character to be tested.

Return value The return value is TRUE if the character is uppercase. Otherwise, it is FALSE.

IsChild

Syntax BOOL IsChild(hWndParent, hWnd)
function IsChild(WndParent, Wnd: HWnd): Bool;

This function indicates whether the window specified by the *hWnd* parameter is a child window or other direct descendant of the window specified by the *hWndParent* parameter. A child window is the direct descendant of a given parent window if that parent window is in the chain of parent windows that leads from the original pop-up window to the child window.

Parameters *hWndParent* **HWND** Identifies a window.

hWnd **HWND** Identifies the window to be checked.

Return value The return value specifies the outcome of the function. It is TRUE if the window identified by the *hWnd* parameter is a child window of the window identified by the *hWndParent* parameter. Otherwise, it is FALSE.

IsClipboardFormatAvailable

Syntax BOOL IsClipboardFormatAvailable(wFormat)
function IsClipboardFormatAvailable(Format: Word): Bool;

This function specifies whether data of a certain type exist in the clipboard.

Parameters *wFormat* **WORD** Specifies a registered clipboard format. For information on clipboard formats, see the description of the **SetClipboardData** function, later in this chapter.

Return value The return value specifies the outcome of the function. It is TRUE if data having the specified format are present. Otherwise, it is FALSE.

Comments This function is typically called during processing of the WM_INITMENU or WM_INITMENUPOPUP message to determine whether the clipboard contains data that the application can paste. If such data are present, the application typically enables the Paste command (in its Edit menu).

IsDialogMessage

Syntax BOOL IsDialogMessage(hDlg, lpMsg)
function IsDialogMessage(Dlg: HWnd; var Msg: TMsg): Bool;

This function determines whether the given message is intended for the modeless dialog box specified by the *hDlg* parameter, and automatically processes the message if it is. When the **IsDialogMessage** function processes a message, it checks for keyboard messages and converts them into selection commands for the corresponding dialog box. For example, the TAB key selects the next control or group of controls, and the DOWN key selects the next control in a group.

If a message is processed by **IsDialogMessage**, it must not be passed to the **Translate-Message** or **DispatchMessage** function. This is because **IsDialogMessage** performs all necessary translating and dispatching of messages.

IsDialogMessage sends WM_GETDLGCODE messages to the dialog function to determine which keys should be processed.

Parameters

hDlg **HWND** Identifies the dialog box.

lpMsg **LPMSG** Points to an **MSG** data structure that contains the message to be checked.

Return value The return value specifies whether or not the given message has been processed. It is nonzero if the message has been processed. Otherwise, it is zero.

Comments Although **IsDialogMessage** is intended for modeless dialog boxes, it can be used with any window that contains controls to provide the same keyboard selection as in a dialog box.

IsDlgButtonChecked

Syntax WORD IsDlgButtonChecked(hDlg, nIDButton)
function IsDlgButtonChecked(Wnd: HWnd; IDButton: Integer): Word;

This function determines whether a button control has a checkmark next to it, and whether a three-state button control is grayed, checked, or neither. The **IsDlgButtonChecked** function sends a BM_GETCHECK message to the button control.

Parameters

hDlg	**HWND** Identifies the dialog box that contains the button control.
nIDButton	**int** Specifies the integer identifier of the button control.

Return value The return value specifies the outcome of the function. It is nonzero if the given control has a checkmark next to it. Otherwise, it is zero. For three-state buttons, the return value is 2 if the button is grayed, 1 if the button has a checkmark next to it, and zero otherwise.

IsIconic

Syntax BOOL IsIconic(hWnd)
function IsIconic(Wnd: HWnd): Bool;

This function specifies whether a window is minimized (iconic).

Parameters

hWnd	**HWND** Identifies the window.

Return value The return value specifies whether the window is minimized. It is nonzero if the window is minimized. Otherwise, it is zero.

IsRectEmpty

Syntax BOOL IsRectEmpty(lpRect)
function IsRectEmpty(var Rect: TRect): Bool;

This function determines whether or not the specified rectangle is empty. A rectangle is empty if the width and/or height are zero.

Parameters *lpRect* **LPRECT** Points to a **RECT** data structure that contains the specified rectangle.

Return value The return value specifies whether or not the given rectangle is empty. It is nonzero if the rectangle is empty. It is zero if the rectangle is not empty.

IsWindow

Syntax BOOL IsWindow(hWnd)
function IsWindow(Wnd: HWnd): Bool;

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

Parameters *hWnd* **HWND** Identifies the window.

Return value The return value specifies whether or not the given window is valid. It is nonzero if *hWnd* is a valid window. Otherwise, it is zero.

IsWindowEnabled

Syntax BOOL IsWindowEnabled(hWnd)
function IsWindowEnabled(Wnd: HWnd): Bool;

This function specifies whether the specified window is enabled for mouse and keyboard input.

Parameters *hWnd* **HWND** Identifies the window.

Return value The return value specifies whether or not the given window is enabled. It is nonzero if the window is enabled. Otherwise, it is zero.

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

IsWindowVisible

Syntax BOOL IsWindowVisible(hWnd)
function IsWindowVisible(Wnd: HWnd): Bool;

The **IsWindowVisible** function returns nonzero anytime an application has made a window visible by using the **ShowWindow** function (even if the

specified window is completely covered by another child or pop-up window, the return value is nonzero).

Parameters *hWnd* **HWND** Identifies the window.

Return value The return value specifies whether or not a given window exists on the screen. It is nonzero if the given window exists on the screen. Otherwise, it is zero.

IsZoomed

Syntax BOOL IsZoomed(hWnd)
function IsZoomed(Wnd: HWnd): Bool;

This function determines whether or not a window has been maximized.

Parameters *hWnd* **HWND** Identifies the window.

Return value The return value specifies whether or not the given window is maximized. It is nonzero if the window is maximized. Otherwise, it is zero.

KillTimer

Syntax BOOL KillTimer(hWnd, nIDEvent)
function KillTimer(Wnd: HWnd; IDEvent: Integer): Bool;

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

Parameters

hWnd **HWND** Identifies the window associated with the given timer event. This must be the same value passed as the *hWnd* parameter to the **SetTimer** function call that created the timer event.

nIDEvent **int** Specifies the timer event to be killed. If the application called **SetTimer** with the *hWnd* parameter set to NULL, this must be the event identifier returned by **SetTimer**. If the *hWnd* parameter of **SetTimer** was a valid window handle, *nIDEvent* must be the value of the *nIDEvent* parameter passed to **SetTimer**.

Return value The return value specifies the outcome of the function. It is nonzero if the event was killed. It is zero if the **KillTimer** function could not find the specified timer event.

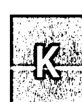
_lclose

Syntax int _lclose(hFile)
function _lclose(FileHandle: Integer): Integer;

This function closes the file specified by the *hFile* parameter. As a result, the file is no longer available for reading or writing.

The *hFile* argument is returned by the call that created or last opened the file.

Value	Meaning
hFile	**int** Specifies the MS-DOS file handle of the file to be closed.



Return value The return value indicates whether the function successfully closed the file. It is zero if the function closed the file, or –1 if the function failed.

_lcreat

Syntax int _lcreat(lpPathName, iAttribute)
function _lcreat(PathName: PChar; Attribute: Integer): Integer;

This function opens a file with the name specified by the *lpPathName* parameter. The *iAttribute* parameter specifies the attributes of the file when the function opens it. If the file does not exist, the function creates a new file and opens it for writing. If the file does exist, the function truncates the file size to zero and opens it for reading and writing. When the function opens the file, the pointer is set to the beginning of the file.

Parameters

lpPathName **LPSTR** Points to a null-terminated character string that names the file to be opened. The string must consist of characters from the ANSI character set.

iAttribute **int** Specifies the file attributes. The parameter must be one of these values:

Value	Meaning
0	Normal; can be read or written without restriction.
1	Read-only; cannot be opened for write; a file with the same name cannot be created.

2	Hidden; not found by directory search.
3	System; not found by directory search.

Return value The return value specifies an MS-DOS file handle if the function was successful. Otherwise, the return value is –1.

LimitEmsPages

Syntax void LimitEmsPages (dwKbytes)
procedure LimitEmsPages(Kbytes: Longint);

This function limits the amount of expanded memory that Windows will assign to an application. It does not limit the amount of expanded memory that the application can get by directly calling INT 67H.

Parameters *dwKbytes* **DWORD** Specifies the number of kilobytes of expanded memory to which the application is to have access.

Return value None.

Comments **LimitEmsPages** has an effect only if expanded memory is installed and being used by Windows. If Windows is not using expanded memory, then the function has no effect.

LineDDA

Syntax void LineDDA(X1, Y1, X2, Y2, lpLineFunc, lpData)
procedure LineDDA(X1, Y1, X2, Y2: Integer; LineFunc: TFarProc; Data: Pointer);

This function computes all successive points in a line starting at the point specified by the *X1* and *Y1* parameters and ending at the point specified by the *X2* and *Y2* parameters. The endpoint is not included as part of the line. For each point on the line, the **LineDDA** function calls the application-supplied function pointed to by the *lpLineFunc* parameter, passing to the function the coordinates of the current point and the *lpData* parameter.

Parameters

X1	**int** Specifies the logical *x*-coordinate of the first point.
Y1	**int** Specifies the logical *y*-coordinate of the first point.
X2	**int** Specifies the logical *x*-coordinate of the endpoint.
Y2	**int** Specifies the logical *y*-coordinate of the endpoint.

lpLineFunc	**FARPROC** Is the procedure-instance address of the application-supplied function. See the following "Comments" section for details.
lpData	**LPSTR** Points to the application-supplied data.

Return value None.

Comments The address passed by the *lpLineFunc* parameter must be created by using the **MakeProcInstance** function.

The callback function must use the Pascal calling convention and must be declared **FAR**.

Callback function

```
void FAR PASCAL LineFunc(X, Y, lpData)
int  X;
int  Y;
LPSTR lpData;
```

LineFunc is a placeholder for the application-supplied function name. The actual name must be exported by including it in an **EXPORTS** statement in the application's module-definition file.

Parameters

X	Specifies the *x*-coordinate of the current point.
Y	Specifies the *y*-coordinate of the current point.
lpData	Points to the application-supplied data.

Return value The function can perform any task. It has no return value.

LineTo

Syntax

```
BOOL LineTo(hDC, X, Y)
function LineTo(DC: HDC; X, Y: Integer): Bool;
```

This function draws a line from the current position up to, but not including, the point specified by the *X* and *Y* parameters. The line is drawn with the selected pen. If no error occurs, the position is set to (*X*,*Y*).

Parameters

hDC	**HDC** Identifies the device context.
X	**int** Specifies the logical *x*-coordinate of the endpoint for the line.

Y	**int** Specifies the logical *y*-coordinate of the endpoint for the line.

Return value The return value specifies whether or not the line is drawn. It is nonzero if the line is drawn. Otherwise, it is zero.

_llseek

Syntax LONG _llseek(hFile, lOffset, iOrigin)
function _llseek(FileHandle: Integer; Offset: Longint; Origin: Integer): Longint;

This function repositions the pointer in a previously opened file. The *iOrigin* parameter specifies the starting position in the file, and *lOffset* specifies how far (in bytes) the function is to move the pointer.

Parameters

hFile	**int** Specifies the MS-DOS file handle of the file.
lOffset	**LONG** Specifies the number of bytes the pointer is to be moved.
iOrigin	**int** Specifies the starting position and direction of the pointer. The parameter must be one of the following values:

Value	Meaning
0	Move the file pointer *lOffset* bytes from the beginning of the file.
1	Move the file pointer *lOffset* bytes from the current position of the file.
2	Move the file pointer *lOffset* bytes from the end of the file.

Return value The return value specifies the new offset of the pointer (in bytes) from the beginning of the file. The return value is –1 if the function fails.

Comments When a file is initially opened, the file pointer is positioned at the beginning of the file. The **_llseek** function permits random access to a file's contents by moving the pointer an arbitrary amount without reading data.

LoadAccelerators

Syntax HANDLE LoadAccelerators(hInstance, lpTableName)
function LoadAccelerators(Instance: THandle; TableName: PChar): THandle;

This function loads the accelerator table named by the *lpTableName* parameter from the executable file associated with the module specified by the *hInstance* parameter.

The **LoadAccelerators** function loads the table only if it has not been previously loaded. Otherwise, it retrieves a handle to the loaded table.

Parameters

hInstance — **HANDLE** Identifies an instance of the module whose executable file contains the accelerator table.

lpTableName — **LPSTR** Points to a string that names the accelerator table. The string must be a null-terminated character string.

Return value The return value identifies the loaded accelerator table if the function is successful. Otherwise, it is NULL.

LoadBitmap

Syntax

HBITMAP LoadBitmap(hInstance, lpBitmapName)
function LoadBitmap(Instance: THandle; BitmapName: PChar): HBitmap;

This function loads the bitmap resource named by the *lpBitmapName* parameter from the executable file associated with the module specified by the *hInstance* parameter.

Parameters

hInstance — **HANDLE** Identifies the instance of the module whose executable file contains the bitmap.

lpBitmapName — **LPSTR** Points to a character string that names the bitmap. The string must be a null-terminated character string.

Return value The return value identifies the specified bitmap. It is NULL if no such bitmap exists.

Comments The application must call the **DeleteObject** function to delete each bitmap handle returned by the **LoadBitmap** function. This also applies to the predefined bitmaps described in the following paragraph.

The **LoadBitmap** function can also be used to access the predefined bitmaps used by Windows. The *hInstance* parameter must be set to NULL, and the *lpBitmapName* parameter must be one of the following values:

- OBM_BTNCORNERS
- OBM_BTSIZE
- OBM_CHECK
- OBM_CHECKBOXES
- OBM_CLOSE
- OBM_COMBO

- OBM_DNARROW
- OBM_DNARROWD
- OBM_LFARROW
- OBM_LFARROWD
- OBM_MNARROW
- OBM_OLD_CLOSE
- OBM_OLD_DNARROW
- OBM_OLD_LFARROW
- OBM_OLD_REDUCE
- OBM_OLD_RESTORE
- OBM_OLD_RGARROW
- OBM_OLD_UPARROW
- OBM_OLD_ZOOM
- OBM_REDUCE
- OBM_REDUCED
- OBM_RESTORE
- OBM_RESTORED
- OBM_RGARROW
- OBM_RGARROWD
- OBM_SIZE
- OBM_UPARROW
- OBM_UPARROWD
- OBM_ZOOM
- OBM_ZOOMD

Bitmap names that begin OBM_OLD represent bitmaps used by Windows versions prior to 3.0.

The *lpBitmapName* parameter can also be a value created by the **MAKEINTRESOURCE** macro. If it is, the ID must reside in the low-order word of *lpBitmapName*, and the high-order word must contain zeros.

LoadCursor

Syntax HCURSOR LoadCursor(hInstance, lpCursorName)
function LoadCursor(Instance: THandle; CursorName: PChar): HCursor;

This function loads the cursor resource named by the *lpCursorName* parameter from the executable file associated with the module specified by the *hInstance* parameter. The function loads the cursor into memory only if it has not been previously loaded. Otherwise, it retrieves a handle to the existing resource.

Parameters

hInstance	**HANDLE** Identifies an instance of the module whose executable file contains the cursor.
lpCursorName	**LPSTR** Points to a character string that names the cursor resource. The string must be a null-terminated character string.

Return value The return value identifies the newly loaded cursor if the function is successful. Otherwise, it is NULL.

Comments The **LoadCursor** function returns a valid cursor handle only if the *lpCursorName* parameter identifies a cursor resource. If *lpCursorName* identifies any type of resource other than a cursor (such as an icon), the return value will not be NULL, even though it is not a valid cursor handle.

Use the **LoadCursor** function to access the predefined cursors used by Windows. To do this, the *hInstance* parameter must be set to NULL, and the *lpCursorName* parameter must be one of the following values:

Value	Meaning
IDC_ARROW	Standard arrow cursor.
IDC_CROSS	Crosshair cursor.
IDC_IBEAM	Text I-beam cursor.
IDC_ICON	Empty icon.
IDC_SIZE	Loads a square with a smaller square inside its lower-right corner.
IDC_SIZENESW	Double-pointed cursor with arrows pointing northeast and southwest.
IDC_SIZENS	Double-pointed cursor with arrows pointing north and south.
IDC_SIZENWSE	Double-pointed cursor with arrows pointing northwest and southeast.
IDC_SIZEWE	Double-pointed cursor with arrows pointing west and east.
IDC_UPARROW	Vertical arrow cursor.
IDC_WAIT	Hourglass cursor.

The *lpCursorName* parameter can contain a value created by the **MAKEINTRESOURCE** macro. If it does, the ID must reside in the low-order word of *lpCursorName*, and the high-order word must be set to zero.

LoadIcon

Syntax HICON LoadIcon(hInstance, lpIconName)
function LoadIcon(Instance: THandle; IconName: PChar): HIcon;

This function loads the icon resource named by the *lpIconName* parameter from the executable file associated with the module specified by the *hInstance* parameter. The function loads the icon only if it has not been previously loaded. Otherwise, it retrieves a handle to the loaded resource.

Parameters

hInstance **HANDLE** Identifies an instance of the module whose executable file contains the icon.

lpIconName **LPSTR** Points to a character string that names the icon resource. The string must be a null-terminated character string.

Return value The return value identifies an icon resource if the function is successful. Otherwise, it is NULL.

Comments Use the **LoadIcon** function to access the predefined icons used by Windows. To do this, the *hInstance* parameter must be set to NULL, and the *lpIconName* parameter must be one of the following values:

Value	Meaning
IDI_APPLICATION	Default application icon.
IDI_ASTERISK	Asterisk (used in informative messages).
IDI_EXCLAMATION	Exclamation point (used in warning messages).
IDI_HAND	Hand-shaped icon (used in serious warning messages).
IDI_QUESTION	Question mark (used in prompting messages).

The *lpIconName* parameter can also contain a value created by the **MAKEINTRESOURCE** macro. If it does, the ID must reside in the low-order word of *lpIconName*, and the high-order word must be set to zero.

LoadLibrary

Syntax HANDLE LoadLibrary(lpLibFileName)
function LoadLibrary(LibFileName: PChar): THandle;

This function loads the library module contained in the specified file and retrieves a handle to the loaded module instance.

Parameters

lpLibFileName **LPSTR** Points to a string that names the library file. The string must be a null-terminated character string.

Return value The return value identifies the instance of the loaded library module. Otherwise, it is a value less than 32 that specifies the error. The following list describes the error values returned by this function:

Value	Meaning
0	Out of memory.
2	File not found.
3	Path not found.
5	Attempt to dynamically link to a task.
6	Library requires separate data segments for each task.
10	Incorrect Windows version.
11	Invalid .EXE file (non-Windows .EXE or error in .EXE image).
12	OS/2 application.
13	DOS 4.0 application.
14	Unknown .EXE type.
15	Attempt in protected (standard or 386 enhanced) mode to load an .EXE created for an earlier version of Windows.
16	Attempt to load a second instance of an .EXE containing multiple, writeable data segments.
17	Attempt in large-frame EMS mode to load a second instance of an application that links to certain nonshareable DLLs already in use.
18	Attempt in real mode to load an application marked for protected mode only.

LoadMenu

Syntax HMENU LoadMenu(hInstance, lpMenuName)
function LoadMenu(Instance: THandle; MenuName: PChar): HMenu;

This function loads the menu resource named by the *lpMenuName* parameter from the executable file associated with the module specified by the *hInstance* parameter.

Parameters

hInstance **HANDLE** Identifies an instance of the module whose executable file contains the menu.

lpMenuName **LPSTR** Points to a character string that names the menu resource. The string must be a null-terminated character string.

Return value The return value identifies a menu resource if the function is successful. Otherwise, it is NULL.

Comments The *lpMenuName* parameter can contain a value created by the **MAKEINTRESOURCE** macro. If it does, the ID must reside in the low-order word of *lpMenuName*, and the high-order word must be set to zero.

LoadMenuIndirect

Syntax HMENU LoadMenuIndirect(lpMenuTemplate)
function LoadMenuIndirect(MenuTemplate: Pointer): HMenu;

This function loads into memory the menu named by the *lpMenuTemplate* parameter. The template specified by *lpMenuTemplate* is a header followed by a collection of one or more **MENUITEMTEMPLATE** structures, each of which may contain one or more menu items and pop-up menus.

Parameters *lpMenuTemplate* **LPSTR** Points to a menu template (which is a collection of one or more **MENUITEMTEMPLATE** structures).

Return value The return value identifies the menu if the function is successful. Otherwise, it is NULL.

LoadModule 3.0

Syntax HANDLE LoadModule(lpModuleName, lpParameterBlock)
function LoadModule(ModuleName: PChar; ParameterBlock: Pointer): THandle;

This function loads and executes a Windows program or creates a new instance of an existing Windows program.

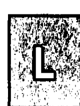
Parameters *lpModuleName* **LPSTR** Points to a null-terminated string that contains the filename of the application to be run. If the *lpModuleName* string does not contain a directory path, Windows will search for the executable file in this order:

1. The current directory.
2. The Windows directory (the directory containing WIN.COM); the **GetWindowsDirectory** function obtains the pathname of this directory.
3. The Windows system directory (the directory containing such system files as KERNEL.EXE); the **GetSystemDirectory** function obtains the pathname of this directory.
4. The directories listed in the PATH environment variable.
5. The list of directories mapped in a network. If the application filename does not contain an extension, then .EXE is assumed.

lpParameterBlock **LPVOID** Points to a data structure consisting of four fields that defines a parameter block. This data structure consists of the following fields:

Field	Type/Description
wEnvSeg	**WORD** Specifies the segment address of the environment under which the module is to run; 0 indicates that the Windows environment is to be copied.
lpCmdLine	**LPSTR** Points to a null-terminated character string that contains a correctly formed command line. This string must not exceed 120 bytes in length.
lpCmdShow	**LPVOID** Points to a data structure containing two **WORD**-length values. The first value must always be set to two. The second value specifies how the application window is to be shown. See the description of the *nCmdShow* paramter of the **ShowWindow** function for a list of the acceptable values.
dwReserved	**DWORD** Is reserved and must be NULL.

All unused fields should be set to NULL, except for *lpCmdLine*, which must point to a null string if it is not used.



Return value The return value identifies the instance of the loaded module if the function was successful. Otherwise, it is a value less than 32 that specifies the error. The following list describes the error values returned by this function:

Value	Meaning
0	Out of memory.
2	File not found.
3	Path not found.
5	Attempt to dynamically link to a task.
6	Library requires separate data segments for each task.
10	Incorrect Windows version.
11	Invalid .EXE file (non-Windows .EXE or error in .EXE image).
12	OS/2 application.
13	DOS 4.0 application.
14	Unknown .EXE type.

15	Attempt in protected (standard or 386 enhanced) mode to load an .EXE created for an earlier version of Windows.
16	Attempt to load a second instance of an .EXE containing multiple, writeable data segments.
17	Attempt in large-frame EMS mode to load a second instance of an application that links to certain nonshareable DLLs already in use.
18	Attempt in real mode to load an application marked for protected mode only.

Comments The **WinExec** function provides an alternative method for executing a program.

LoadResource

Syntax HANDLE LoadResource(hInstance, hResInfo)
function LoadResource(Instance, ResInfo: THandle): THandle;

This function loads a resource identified by the *hResInfo* parameter from the executable file associated with the module specified by the *hInstance* parameter. The function loads the resource into memory only if it has not been previously loaded. Otherwise, it retrieves a handle to the existing resource.

Parameters

hInstance **HANDLE** Identifies an instance of the module whose executable file contains the resource.

hResInfo **HANDLE** Identifies the desired resource. This handle is assumed to have been created by using the **FindResource** function.

Return value The return value identifies the global memory block to receive the data associated with the resource. It is NULL if no such resource exists.

Comments The resource is not actually loaded until the **LockResource** function is called to translate the handle returned by **LoadResource** into a far pointer to the resource data.

LoadString

Syntax int LoadString(hInstance, wID, lpBuffer, nBufferMax)
function LoadString(Instance: THandle; ID: Word; Buffer: PChar; BufferMax: Integer): Integer;

This function loads a string resource identified by the *wID* parameter from the executable file associated with the module specified by the *hInstance*

parameter. The function copies the string into the buffer pointed to by the *lpBuffer* parameter, and appends a terminating null character.

Parameters

hInstance	**HANDLE** Identifies an instance of the module whose executable file contains the string resource.
wID	**WORD** Specifies the integer identifier of the string to be loaded.
lpBuffer	**LPSTR** Points to the buffer that receives the string.
nBufferMax	**int** Specifies the maximum number of characters to be copied to the buffer. The string is truncated if it is longer than the number of characters specified.

Return value The return value specifies the actual number of characters copied into the buffer. It is zero if the string resource does not exist.

LOBYTE

Syntax

BYTE LOBYTE(nInteger)
function LoByte(A: Word): Byte;

This macro extracts the low-order byte from the short-integer value specified by the *nInteger* parameter.

Parameters *nInteger* **int** Specifies the value to be converted.

Return value The return value specifies the low-order byte of the value.

LocalAlloc

Syntax

HANDLE LocalAlloc(wFlags, wBytes)
function LocalAlloc(Flags, Bytes: Word): THandle;

This function allocates the number of bytes of memory specified by the *wBytes* parameter from the local heap. The memory block can be either fixed or moveable, as specified by the *wFlags* parameter.

Parameters *wFlags* **WORD** Specifies how to allocate memory. It can be one or more of the following values:

Value	Meaning
LMEM_DISCARDABLE	Allocates discardable memory. Can only be used with LMEM_MOVEABLE.

LMEM_FIXED	Allocates fixed memory.
LMEM_MODIFY	Modifies the LMEM_DISCARDABLE flag. Can only be used with LMEM_DISCARDABLE.
LMEM_MOVEABLE	Allocates moveable memory. Cannot be used with LMEM_FIXED.
LMEM_NOCOMPACT	Does not compact or discard memory to satisfy the allocation request.
LMEM_NODISCARD	Does not discard memory to satisfy the allocation request.
LMEM_ZEROINIT	Initializes memory contents to zero.

Choose LMEM_FIXED or LMEM_MOVEABLE, and then combine others as needed by using the bitwise OR operator.

wBytes **WORD** Specifies the total number of bytes to be allocated.

Return value The return value identifies the newly allocated local memory block if the function is successful. Otherwise, it is NULL.

Comments If the data segment that contains the heap is moveable, calling this function will cause the data segment to move if Windows needs to increase the size of the heap and cannot increase the size of the heap in its current location. An application can prevent Windows from moving the data segment by calling the **LockData** function to lock the data segment.

If this function is successful, it allocates at least the amount requested. The actual amount allocated may be greater. To determine the actual amount allocated, call the **LocalSize** function.

LocalCompact

Syntax WORD LocalCompact(wMinFree)
function LocalCompact(MinFree: Word): Word;

This function generates the number of free bytes of memory specified by the *wMinFree* parameter by compacting, if necessary, the module's local heap. The function checks the local heap for the specified number of contiguous free bytes. If the bytes do not exist, the **LocalCompact** function compacts local memory by first moving all unlocked moveable blocks into high memory. If this does not generate the requested amount of space, the

function discards moveable and discardable blocks that are not locked down, until the requested amount of space is generated, whenever possible.

Parameters *wMinFree* **WORD** Specifies the number of free bytes desired. If *wMinFree* is zero, the function returns a value but does not compact memory.

Return value The return value specifies the number of bytes in the largest block of free local memory.

LocalDiscard

Syntax HANDLE LocalDiscard(hMem)
function LocalDiscard(Mem: THandle): THandle;

This function discards the local memory block specified by the *hMem* parameter. The lock count of the memory block must be zero.

The local memory block is removed from memory, but its handle remains valid. An application can subsequently pass the handle to the **LocalReAlloc** function to allocate another local memory block identified by the same handle.

Parameters *hMem* **HANDLE** Identifies the local memory block to be discarded.

Return value The return value specifies the outcome of the function. It is NULL if the function is successful. Otherwise, it is equal to *hMem*.

LocalFlags

Syntax WORD LocalFlags(hMem)
function LocalFlags(Mem: THandle): Word;

This function returns information about the specified local memory block.

Parameters *hMem* **HANDLE** Identifies the local memory block.

Return value The return value contains one of the following memory-allocation flags in the high byte:

Value	Meaning
LMEM_DISCARDABLE	Block is marked as discardable.
LMEM_DISCARDED	Block has been discarded.

The low byte of the return value contains the reference count of the block. Use the LMEM_LOCKCOUNT mask to retrieve the lock-count value from the return value.

LocalFree

Syntax HANDLE LocalFree(hMem)
function LocalFree(Mem: THandle): THandle;

This function frees the local memory block identified by the *hMem* parameter and invalidates the handle of the memory block.

Parameters *hMem* **HANDLE** Identifies the local memory block to be freed.

Return value The return value specifies the outcome of the function. It is NULL if the function is successful. Otherwise, it is equal to *hMem*.

LocalHandle

Syntax HANDLE LocalHandle(wMem)
function LocalHandle(Mem: Word): THandle;

This function retrieves the handle of the local memory object whose address is specified by the *wMem* parameter.

Parameters *wMem* **WORD** Specifies the address of a local memory object.

Return value The return value identifies the local memory object.

LocalInit

Syntax BOOL LocalInit(wSegment, pStart, pEnd)
function LocalInit(Segment, Start, EndPos: Word): Bool;

This function initializes a local heap in the segment specified by the *wSegment* parameter.

Parameters

wSegment **WORD** Specifies the segment address of the segment that is to contain the local heap.

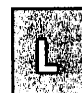
pStart **PSTR** Specifies the address of the start of the local heap within the segment.

pEnd **PSTR** Specifies the address of the end of the local heap within the segment.

Return value The return value specifies a Boolean value that is nonzero if the heap is initialized. Otherwise, it is zero.

Comments If the *pStart* parameter is zero, the *pEnd* parameter specifies the offset of the last byte of the global heap from the end of the segment. For example, to initialize a 4096-byte heap with the first byte at byte 0, set *pStart* to 0 and *pEnd* to 4095. **LocalInit** calls the **GlobalLock** function for the data segment that contains the local heap. This ensures that the data segment will not be moved in memory. However, the memory will be moved if both of these conditions are true:

1. The data segment is moveable.
2. The application calls the **LocalAlloc** or **LocalReAlloc** function and, as a result, Windows must increase the size of the heap. If Windows cannot increase the size of the data segment that contains the local heap without moving it, Windows will move the data segment.



An application can explicitly prevent Windows from moving the data segment by calling the **LockData** function to lock the data segment.

An application can remove this initial lock count by calling the **UnlockData** function.

LocalLock

Syntax PSTR LocalLock(hMem)
function LocalLock(Mem: THandle): Pointer;

This function locks the local memory block specified by the *hMem* parameter. The block is locked into memory at the given address and its reference count is increased by one. Locked memory cannot be moved or discarded. The block remains locked in memory until its reference count is decreased to zero by using the **LocalUnlock** function.

Parameters *hMem* **HANDLE** Identifies the local memory block to be locked.

Return value The return value points to the first byte of memory in the local block if the function is successful. Otherwise, it is NULL.

LocalReAlloc

Syntax HANDLE LocalReAlloc(hMem, wBytes, wFlags)

function LocalReAlloc(Mem: THandle; Bytes, Flags: Word): THandle;

This function changes the size of the local memory block specified by the *hMem* parameter by increasing or decreasing its size to the number of bytes specified by the *wBytes* parameter, or changes the attributes of the specified memory block.

Parameters

hMem **HANDLE** Identifies the local memory block to be reallocated.

wBytes **WORD** Specifies the new size of the memory block.

wFlags **WORD** Specifies how to reallocate the local memory block. It can be one or more of the following values:

Value	**Meaning**
LMEM_DISCARDABLE	Memory is discardable. This flag can only be used with LMEM_MODIFY.
LMEM_MODIFY	Memory flags are modified. The *wBytes* parameter is ignored. This flag can only be used with LMEM_DISCARDABLE.
LMEM_MOVEABLE	Memory is moveable. If *wBytes* is zero, this flag causes a previously fixed block to be freed or a previously moveable object to be discarded (if the block's reference count is zero). If *wBytes* is nonzero and the block specified by *hMem* is fixed, this flag allows the reallocated block to be moved to a new fixed location. (Note that the handle returned by the **LocalReAlloc** function in this case may be different from the handle passed to the function.) This flag cannot be used with LMEM_MODIFY.
LMEM_NOCOMPACT	Memory will not be compacted or discarded to satisfy the allocation request. This flag cannot be used with LMEM_MODIFY.

LMEM_NODISCARD	Objects will not be discarded to satisfy the allocation request. This flag cannot be used with LMEM_MODIFY.
LMEM_ZEROINIT	If the block is growing, the additional memory contents are initialized to zero. This flag cannot be used with LMEM_MODIFY.

Return value The return value identifies the reallocated local memory if the function is successful. It is NULL if the local memory block cannot be reallocated.

The return value is always identical to the *hMem* parameter, unless the LMEM_MOVEABLE flag is used to allow movement of a fixed block of memory to a new fixed location.

Comments If the data segment that contains the heap is moveable, calling this function will cause the data segment to move if Windows must increase the size of the heap and cannot increase the size of the heap in its current location. An application can prevent Windows from moving the data segment by calling the **LockData** function to lock the data segment.

LocalShrink

Syntax WORD LocalShrink(hSeg, wSize)
function LocalShrink(Seg: THandle; Size: Word): Word;

This function shrinks the specified heap to the size specified by the *wSize* parameter. The minimum size for the automatic local heap is defined in the application's module definition file.

Parameters

hSeg	**HANDLE** Identifies the segment that contains the local heap.
wSize	**WORD** Specifies the size (in bytes) desired for the local heap after shrinkage.

Return value The return value specifies the size of the local heap after shrinkage.

Comments If *hSeg* is zero, the **LocalShrink** function reduces the local heap in the current data segment. Windows will not shrink that portion of the data segment that contains the stack and the static variables.

Use the **GlobalSize** function to determine the new size of the data segment.

LocalSize

Syntax WORD LocalSize(hMem)
function LocalSize(Mem: THandle): Word;

This function retrieves the current size (in bytes) of the local memory block specified by the *hMem* parameter.

Parameters *hMem* **HANDLE** Identifies the local memory block.

Return value The return value specifies the size (in bytes) of the specified memory block. It is NULL if the given handle is not valid.

Comments The actual size of a memory block sometimes is larger than the size requested when the memory was allocated.

LocalUnlock

Syntax BOOL LocalUnlock(hMem)
function LocalUnlock(Mem: THandle): Bool;

This function unlocks the local memory block specified by the *hMem* parameter and decreases the block's reference count by one. The block is completely unlocked, and subject to moving or discarding, if the reference count is decreased to zero.

Parameters *hMem* **HANDLE** Identifies the local memory block to be unlocked.

Return value The return value is zero if the block's reference count was decreased to zero. Otherwise, the return value is nonzero.

LockData

Syntax HANDLE LockData(Dummy)
function LockData(Dummy: Integer): THandle;

This macro locks the current data segment in memory. It is intended to be used in modules that have moveable data segments.

Parameters *Dummy* **int** Is not used. It should be set to zero.

Return value The return value identifies the locked data segment if the function is successful. Otherwise, it is NULL.

LockResource

Syntax LPSTR LockResource(hResData)
function LockResource(ResData: THandle): Pointer;

This function retrieves the absolute memory address of the loaded resource identified by the *hResData* parameter. The resource is locked in memory and the given address and its reference count are increased by one. The locked resource is not subject to moving or discarding.

The resource remains locked in memory until its reference count is decreased to zero through calls to the **FreeResource** function.

If the resource identified by *hResData* has been discarded, the resource-handler function (if any) associated with the resource is called before the **LockResource** function returns. The resource-handler function can recalculate and reload the resource if desired. After the resource-handler function returns, **LockResource** makes another attempt to lock the resource and returns with the result.

Parameters *hResData* **HANDLE** Identifies the desired resource. This handle is assumed to have been created by using the **LoadResource** function.

Return value The return value points to the first byte of the loaded resource if the resource was locked. Otherwise, it is NULL.

Comments Using the handle returned by the **FindResource** function for the *hResData* parameter causes an error.

Use the **UnlockResource** macro to unlock a resource that was locked by using **LockResource**.

LockSegment

Syntax HANDLE LockSegment(wSegment)
function LockSegment(Segment: Word): THandle;

This function locks the segment whose segment address is specified by the *wSegment* parameter. If *wSegment* is –1, the **LockSegment** function locks the current data segment.

Except for nondiscardable segments in protected (standard or 386 enhanced) mode, the segment is locked into memory at the given address

and its lock count is increased by one. Locked memory is not subject to moving or discarding except when a portion of the segment is being reallocated by the **GlobalReAlloc** function. The segment remains locked in memory until its lock count is decreased to zero.

In protected mode, **LockSegment** increments the lock count of discardable and automatic data segments only.

Each time an application calls **LockSegment** for a segment, it must eventually call **UnlockSegment** for the segment. The **UnlockSegment** function decreases the lock count for the segment. Other functions also can affect the lock count of a memory object. See the description of the **GlobalFlags** function for a list of the functions that affect the lock count.

Parameters

wSegment — **WORD** Specifies the segment address of the segment to be locked. If *wSegment* is –1, the **LockSegment** function locks the current data segment.

Return value

The return value identifies the data segment if the function is successful. If the object has been discarded or an error occurs, the return value is NULL.

_lopen

Syntax

int _lopen(lpPathName, iReadWrite)
function _lopen(PathName: PChar; ReadWrite: Integer): Integer;

This function opens the file with the name specified by the *lpPathName* parameter. The *iReadWrite* parameter specifies the access mode of the file when the function opens it. When the function opens the file, the pointer is set to the beginning of the file.

Parameters

lpPathName — **LPSTR** Points to a null-terminated character string that names the file to be opened. The string must consist of characters from the ANSI character set.

iReadWrite — **int** Specifies whether the function is to open the file with read access, write access, or both. The parameter must be one of the following values:

Value	Meaning
OF_READ	Opens the file for reading only.
OF_READWRITE	Opens the file for reading and writing.

OF_SHARE_COMPAT	Opens the file with compatibility mode, allowing any process on a given machine to open the file any number of times. **OpenFile** fails if the file has been opened with any of the other sharing modes.
OF_SHARE_DENY_NONE	Opens the file without denying other processes read or write access to the file. **OpenFile** fails if the file has been opened in compatibility mode by any other process.
OF_SHARE_DENY_READ	Opens the file and denies other processes read access to the file. **OpenFile** fails if the file has been opened in compatibility mode or for read access by any other process.
OF_SHARE_DENY_WRITE	Opens the file and denies other processes write access to the file. **OpenFile** fails if the file has been opened in compatibility or for write access by any other process.
OF_SHARE_EXCLUSIVE	Opens the file with exclusive mode, denying other processes both read and write access to the file. **OpenFile** fails if the file has been opened in any other mode for read or write access, even by the current process.
OF_WRITE	Opens the file for writing only.

Return value The return value specifies an MS-DOS file handle if the function opened the file. Otherwise, it is –1.

LOWORD

Syntax WORD LOWORD(dwInteger)
function LoWord(A: Longint): Word;

This macro extracts the low-order word from the 32-bit integer value specified by the *dwInteger* parameter.

Parameters *dwInteger* **DWORD** Specifies the value to be converted.

Return value The return value specifies the low-order word of the 32-bit integer value.

LPtoDP

Syntax BOOL LPtoDP(hDC, lpPoints, nCount)
function LPtoDP(DC: HDC; var Points; Count: Integer): Bool;

This function converts logical points into device points. The **LPtoDP** function maps the coordinates of each point specified by the *lpPoints* parameter from GDI's logical coordinate system into a device coordinate system. The conversion depends on the current mapping mode.

Parameters

hDC **HANDLE** Identifies the device context.

lpPoints **LPPOINT** Points to an array of points. Each point in the array is a **POINT** data structure.

nCount **int** Specifies the number of points in the array.

Return value The return value specifies whether or not all points are converted. It is nonzero if all points are converted. Otherwise, it is zero.

_lread

Syntax int _lread(hFile, lpBuffer, wBytes)
function _lread(FileHandle: Integer; Buffer: PChar; Bytes: Integer): Word;

This function reads data from the file identified by the *hFile* parameter. The *wBytes* parameter specifies the number of bytes to read. The function return value indicates the number of bytes actually read. The return value is zero if the function attempted to read the file at EOF.

Parameters

hFile **int** Specifies the MS-DOS file handle of the file to be read.

lpBuffer **LPSTR** Points to a buffer that is to receive the data read from the file.

wBytes **WORD** Specifies the number of bytes to be read from the file.

Return value The return value indicates the number of bytes which the function actually read from the file, or –1 if the function fails. The return value is

less than *wBytes* if the function encountered the end of the file (EOF) before reading the specified number of bytes.

lstrcat

Syntax LPSTR lstrcat(lpString1, lpString2)
function lstrcat(Str1, Str2: PChar): PChar;

This function concatenates *lpString2* to the string specified by *lpString1*, terminates the resulting string with a null character, and returns a far pointer to the concatenated string *(lpString1)*.

All strings must be less than 64K in size.

Parameters

lpString1	**LPSTR** Points to byte array containing a null-terminated string to which the function is to append *lpString2*. The byte array containing the string must be large enough to contain both strings.
lpString2	**LPSTR** Points to the null-terminated string which the function is to append to *lpString1*.

Return value The return value specifies a pointer to *lpString1*. It is zero if the function fails.

lstrcmp 3.0

Syntax int lstrcmp(lpString1, lpString2)
function lstrcmp(Str1, Str2: PChar): Integer;

This function compares the two strings identified by *lpString1* and *lpString2* lexicographically and returns a value indicating their relationship. If the strings are otherwise equal, this function uses the case of characters in the string to determine their relationship.

Uppercase characters evaluate lower than lowercase characters. The comparison is made based on the current language selected by the user at setup or with the Control Panel. This function is not equivalent to the **strcmp** C run-time library function.

All strings must be less than 64K in size.

Parameters

lpString1	**LPSTR** Points to the first null-terminated string to be compared.

lpString2	**LPSTR** Points to the second null-terminated string to be compared.

Return value The return value indicates whether *lpString1* is less than, equal to, or greater than *lpString2*. This relationship is outlined in the following:

Value	Meaning
< 0	*lpString1* is less than *lpString2*.
= 0	*lpString1* is identical to *lpString2*.
> 0	*lpString1* is greater than *lpString2*.

lstrcmpi 3.0

Syntax int lstrcmpi(lpString1, lpString2)
function lstrcmpi(Str1, Str2: PChar): Integer;

This function compares the two strings identified by *lpString1* and *lpString2* lexicographically and returns a value indicating their relationship. The comparison is not case-sensitive. The comparison is made based on the current language selected by the user at setup or with the Control Panel. This function is not equivalent to the **strcmpi** C run-time library function.

All strings must be less than 64K in size.

Parameters

lpString1	**LPSTR** Points to the first null-terminated string to be compared.
lpString2	**LPSTR** Points to the second null-terminated string to be compared.

Return value The return value indicates whether *lpString1* is less than, equal to, or greater than *lpString2*. This relationship is outlined in the following table:

Value	Meaning
< 0	*lpString1* is less than *lpString2*.
= 0	*lpString1* is identical to *lpString2*.
> 0	*lpString1* is greater than *lpString2*.

lstrcpy

Syntax LPSTR lstrcpy(lpString1, lpString2)
function lstrcpy(Str1, Str2: PChar): PChar;

This function copies *lpString2*, including the terminating null character, to the location specified by *lpString1*, and returns *lpString1*. All strings must be less than 64K in size.

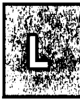
Parameters

lpString1	**LPSTR** Points to a buffer to receive the contents of *lpString2*. The buffer must be large enough to contain *lpString2*.
lpString2	**LPSTR** Points to the null-terminated string to be copied.

Return value The return value specifies a pointer to *lpString1*. It is zero if the function fails.

lstrlen

Syntax int lstrlen(lpString)
function lstrlen(Str: PChar): Integer;



This function returns the length, in bytes, of *lpString*, not including the terminating null character. All strings must be less than 64K in size.

Parameters

lpString	**LPSTR** Points to a null-terminated string.

Return value The return value specifies the length of *lpString*. There is no error return.

_lwrite

Syntax int _lwrite(hFile, lpBuffer, wBytes)
function _lwrite(FileHandle: Integer; Buffer: PChar; Bytes: Integer): Word;

This function writes data into the file specified by the *hFile* parameter. The *wBytes* parameter specifies the number of bytes to write from the buffer identified by *lpBuffer*. The function return value indicates the number of bytes actually written to the file.

Parameters

hFile	**int** Specifies the MS-DOS file handle of the file into which data is to be written.
lpBuffer	**LPSTR** Points to a buffer that contains the data to be written to the file.
wBytes	**WORD** Specifies the number of bytes to be written to the file.

Return value The return value indicates the number of bytes actually written to the file, or –1 if the function fails.

Comments The buffer specified by *lpBuffer* cannot extend past the end of a segment.

MAKEINTATOM

Syntax LPSTR MAKEINTATOM(wInteger)
type MakeIntAtom = Pstr;

This macro creates an integer atom that represents a character string of decimal digits.

Integer atoms created by this macro can be added to the atom table by means of the **AddAtom** function.

Parameters *wInteger* **WORD** Specifies the numeric value of the atom's character string.

Return value The return value points to the atom created for the given integer.

Comments Although the return value of the **MAKEINTATOM** macro is cast as an **LPSTR**, the return value cannot be used as a string pointer, except when passing it to atom-management functions that require an **LPSTR** parameter.

The return value is actually a 32-bit value. The low-order word of this 32-bit value contains the value of the integer specified by the *wInteger* parameter.

The **DeleteAtom** function always succeeds for integer atoms, even though it does nothing, and the **GetAtomName** function always returns the string form of the integer atom.

MAKEINTRESOURCE

Syntax LPSTR MAKEINTRESOURCE (nInteger)
type MakeIntResource = Pstr;

This macro converts an integer value into a long pointer to a string, with the high-order word of the long pointer set to zero.

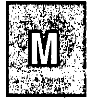
Parameters *nInteger* **int** Specifies the integer value to be converted.

Return value The return value points to a string.

MAKELONG

Syntax DWORD MAKELONG(wLow, wHigh)
function MakeLong(A, B: Word): Longint;

This macro creates an unsigned long integer by concatenating two integer values, specified by the *wLow* and *wHigh* parameters.



Parameters *wLow* **WORD** Specifies the low-order word of the new long value.

wHigh **WORD** Specifies the high-order word of the new long value.

Return value The return value specifies an unsigned long-integer value.

MAKEPOINT

Syntax POINT MAKEPOINT(dwInteger)
type MakePoint = TPoint;

This macro converts a long value that contains the *x*- and *y*-coordinates of a point into a **POINT** data structure.

Parameters *dwInteger* **DWORD** Specifies the *x*- and *y*-coordinates of a point.

Return value The return value specifies the **POINT** data structure.

MakeProcInstance

Syntax FARPROC MakeProcInstance(lpProc, hInstance)
function MakeProcInstance(Proc: TFarProc; Instance: THandle): TFarProc;

This function creates a procedure-instance address. A procedure-instance address points to prolog code that is executed before the function is executed. The prolog binds the data segment of the instance specified by the *hInstance* parameter to the function pointed to by the *lpProc* parameter. When the function is executed, it has access to variables and data in that instance's data segment.

The **FreeProcInstance** function frees the function from the data segment bound to it by the **MakeProcInstance** function.

Parameters

lpProc	**FARPROC** Is a procedure-instance address.
hInstance	**HANDLE** Identifies the instance associated with the desired data segment.

Return value The return value points to the function if the function is successful. Otherwise, it is NULL.

Comments The **MakeProcInstance** function must only be used to access functions from instances of the current module. The function is not required for library modules.

After **MakeProcInstance** has been called for a particular function, all calls to that function should be made through the retrieved address.

MakeProcInstance will create more than one procedure instance. An application should not call **MakeProcInstance** more than once using the same function and instance handle to avoid wasting memory.

To bind a data segment to a function, the function must be exported in the **EXPORTS** statement of the module-definition file.

MapDialogRect

Syntax

```
void MapDialogRect(hDlg, lpRect)
procedure MapDialogRect(Dlg: HWnd; var Rect: TRect);
```

This function converts the dialog-box units given in the *lpRect* parameter to screen units. Dialog-box units are stated in terms of the current dialog base unit derived from the average width and height of characters in the system font. One horizontal unit is one-fourth of the dialog base width unit, and one vertical unit is one-eighth of the dialog base height unit. The **GetDialogBaseUnits** function returns the dialog base units in pixels.

The **MapDialogRect** function replaces the dialog-box units in *lpRect* with screen units (pixels), so that the rectangle can be used to create a dialog box or position a control within a box.

Parameters

hDlg	**HWND** Identifies a dialog box.
lpRect	**LPRECT** Points to a **RECT** data structure that contains the dialog-box coordinates to be converted.

Return value None.

Comments The *hDlg* parameter must be created by using the **CreateDialog** or **DialogBox** function.

MapVirtualKey 3.0

Syntax WORD MapVirtualKey(wCode, wMapType)
function MapVirtualKey(Code, MapType: Word): Word;

This function accepts a virtual-key code or scan code for a key and returns the corresponding scan code, virtual-key code, or ASCII value. The value of the *wMapType* parameter determines the type of mapping which this function performs.

Parameters

wCode **WORD** Specifies the virtual-key code or scan code for a key. The interpretation of the *wCode* parameter depends on the value of the *wMapType* parameter.

wMapType **WORD** Specifies the type of mapping to be performed. The *wMapType* parameter can be any of the following values:

Value	Meaning
0	The *wCode* parameter specifies a virtual-key code, and the function returns the corresponding scan code.
1	The *wCode* parameter specifies a scan code, and the function returns the corresponding virtual-key code.
2	The *wCode* parameter specifies a virtual-key code, and the function returns the corresponding unshifted ASCII value.

Other values are reserved.

Return value The return value depends on the value of the *wCode* and *wMapType* parameters. See the description of the *wMapType* parameter for more information.

max

Syntax int max(value1, value2)

This macro returns the greater of the values contained in the *value1* and *value2* parameters.

Parameters

value1	Specifies the first of two values.
value2	Specifies the second of two values.

Return value The return value specifies *value1* or *value2*, whichever is greater.

Comments The values identified by the *value1* and *value2* parameters can be any ordered type.

MessageBeep

Syntax void MessageBeep(wType)
procedure MessageBeep(BeepType: Word);

This function generates a beep at the system speaker.

Parameters

wType	**WORD** Is not used. It should be set to zero.

Return value None.

MessageBox

Syntax intMessageBox(hWndParent, lpText, lpCaption, wType)
function MessageBox(WndParent: HWnd; Txt, Caption: PChar; TextType: Word): Integer;

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

Parameters

hWndParent	**HWND** Identifies the window that owns the message box.
lpText	**LPSTR** Points to a null-terminated string containing the message to be displayed.
lpCaption	**LPSTR** Points to a null-terminated character string to be used for the dialog-box caption. If the *lpCaption* parameter is NULL, the default caption "Error" is used.

wType **WORD** Specifies the contents of the dialog box. It can be any combination of the values shown in Table 4.11, "Message box types," joined by the bitwise OR operator.

Return value The return value specifies the outcome of the function. It is zero if there is not enough memory to create the message box. Otherwise, it is one of the following menu-item values returned by the dialog box:

Parameters

IDABORT	Abort button pressed.
IDCANCEL	Cancel button pressed.
IDIGNORE	Ignore button pressed.
IDNO	No button pressed.
IDOK	OK button pressed.
IDRETRY	Retry button pressed.
IDYES	Yes button pressed.

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

Comments When a system-modal message box is created to indicate that the system is low on memory, the strings passed as the *lpText* and *lpCaption* parameters should not be taken from a resource file, since an attempt to load the resource may fail.

When an application calls the **MessageBox** function and specifies the MB_ICONHAND and MB_SYSTEMMODAL flags for the *wType* parameter, Windows will display the resulting message box regardless of available memory. When these flags are specified, Windows limits the length of the message-box text to one line.

If a message box is created while a dialog box is present, use the handle of the dialog box as the *hWndParent* parameter. The *hWndParent* parameter should not identify a child window, such as a dialog-box control.

Table 4.11 shows the message box types.

Table 4.11 Message box types

Value	Meaning
MB_ABORTRETRYIGNORE	Message box contains three push buttons: Abort, Retry, and Ignore.
MB_APPLMODAL	The user must respond to the message box before continuing work in the window identified by the *hWndParent* parameter. However, the user can move to the windows of other applications and work in those windows. MB_APPLMODAL is the default if neither MB_SYSTEMMODAL nor MB_TASKMODAL are specified.

Table 4.11: Message box types (continued)

MB_DEFBUTTON1	First button is the default. Note that the first button is always the default unless MB_DEFBUTTON2 or MB_DEFBUTTON3 is specified.
MB_DEFBUTTON2	Second button is the default.
MB_DEFBUTTON3	Third button is the default.
MB_ICONASTERISK	Same as MB_ICONINFORMATION.
MB_ICONEXCLAMATION	An exclamation-point icon appears in the message box.
MB_ICONHAND	Same as MB_ICONSTOP.
MB_ICONINFORMATION	An icon consisting of a lowercase i in a circle appears in the message box.
MB_ICONQUESTION	A question-mark icon appears in the message box.
MB_ICONSTOP	A stop sign icon appears in the message box.
MB_OK	Message box contains one push button: OK.
MB_OKCANCEL	Message box contains two push buttons: OK and Cancel.
MB_RETRYCANCEL	Message box contains two push buttons: Retry and Cancel.
MB_SYSTEMMODAL	All applications are suspended until the user responds to the message box. Unless the application specifies MB_ICONHAND, the message box does not become modal until after it is created; consequently, the parent window and other windows continue to receive messages resulting from its activation. System-modal message boxes are used to notify the user of serious, potentially damaging errors that require immediate attention (for example, running out of memory).
MB_TASKMODAL	Same as MB_APPMODAL except that all the top-level windows belonging to the current task are disabled if the *hWndOwner* parameter is NULL. This flag should be used when the calling application or library does not have a window handle available, but still needs to prevent input to other windows in the current application without suspending other applications.
MB_YESNO	Message box contains two push buttons: Yes and No.
MB_YESNOCANCEL	Message box contains three push buttons: Yes, No, and Cancel.

min

Syntax int min(value1, value2)

This macro returns the lesser of the values specified by the *value1* and *value2* parameters, respectively.

Parameters

value1	Specifies the first of two values.
value2	Specifies the second of two values.

Return value The return value specifies *value1* or *value2*, whichever is less.

Comments The values identified by the *value1* and *value2* parameters can be any ordered type.

ModifyMenu 3.0

Syntax BOOL ModifyMenu(hMenu, nPosition, wFlags, wIDNewItem, lpNewItem)
function ModifyMenu(Menu: HMenu; Position, Flags, IDNewItem: Word; NewItem: PChar): Bool;

This function changes an existing menu item at the position specified by the *nPosition* parameter. The application specifies the new state of the menu item by setting values in the *wFlags* parameter. If this function replaces a pop-up menu associated with the menu item, it destroys the old pop-up menu and frees the memory used by the pop-up menu.

Parameters

hMenu **HMENU** Identifies the menu to be changed.

nPosition **WORD** Specifies the menu item to be changed. The interpretation of the *nPosition* parameter depends upon the setting of the *wFlags* parameter.

IfwFlags is:	**nPosition**
MF_BYPOSITION	Specifies the position of the existing menu item. The first item in the menu is at position zero.
MF_BYCOMMAND	Specifies the command ID of the existing menu item.

wFlags **WORD** Specifies how the *nPosition* parameter is interpreted and information about the changes to be made to the menu item. It consists of one or more values listed in the following "Comments" section.

wIDNewItem **WORD** Specifies either the command ID of the modified menu item or, if *wFlags* is set to MF_POPUP, the menu handle of the pop-up menu.

M

lpNewItem	**LPSTR** Specifies the content of the changed menu item. If *wFlags* is set to MF_STRING (the default), then *lpNewItem* is a long pointer to a null-terminated character string. If *wFlags* is set to MF_BITMAP instead, then *lpNewItem* contains a bitmap handle (**HBITMAP**) in its low-order word. If *wFlags* is set to MF_OWNERDRAW, *lpNewItem* specifies an application-supplied 32-bit value which the application can use to maintain additional data associated with the menu item. This 32-bit value is available to the application in the **itemData** field of the structure, pointed to by the *lParam* parameter of the following messages: WM_MEASUREITEM WM_DRAWITEM These messages are sent when the menu item is initially displayed, or is changed.

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

Comments Whenever a menu changes (whether or not the menu resides in a window that is displayed), the application should call **DrawMenuBar**. In order to change the attributes of existing menu items, it is much faster to use the **CheckMenuItem** and **EnableMenuItem** functions.

Each of the following groups lists flags that should not be used together:

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

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

Parameters

MF_BITMAP	Uses a bitmap as the menu item. The low-order word of the *lpNewItem* parameter contains the handle of the bitmap.
MF_BYCOMMAND	Specifies that the *nPosition* parameter gives the menu item control ID number. This is the default if neither MF_BYCOMMAND nor MF_POSITION is set.

MF_BYPOSITION	Specifies that the *nPosition* parameter gives the position of the menu item to be changed rather than an ID number.
MF_CHECKED	Places a checkmark next to the menu item. If the application has supplied checkmark bitmaps (see **SetMenuItemBitmaps**), setting this flag displays the "checkmark on" bitmap next to the menu item.
MF_DISABLED	Disables the menu item so that it cannot be selected, but does not gray it.
MF_ENABLED	Enables the menu item so that it can be selected and restores it from its grayed state.
MF_GRAYED	Disables the menu item so that it cannot be selected and grays it.
MF_MENUBARBREAK	Same as MF_MENUBREAK except that for pop-up menus, separates the new column from the old column with a vertical line.
MF_MENUBREAK	Places the menu item on a new line for static menu-bar items. For pop-up menus, this flag places the item in a new column, with no dividing line between the columns.
MF_OWNERDRAW	Specifies that the menu item is an owner-draw item. The window that owns the menu receives a WM_MEASUREITEM message when the menu is displayed for the first time to retrieve the height and width of the menu item. The WM_DRAWITEM message is then sent whenever the owner must update the visual appearance of the menu item. This option is not valid for a top-level menu item.
MF_POPUP	Specifies that the item has a pop-up menu associated with it. The *wIDNewItem* parameter specifies a handle to a pop-up menu to be associated with the menu item. Use this flag for adding either a top-level pop-up menu or adding a hierarchical pop-up menu to a pop-up menu item.
MF_SEPARATOR	Draws a horizontal dividing line. You can only use this flag in a pop-up menu. This line cannot be grayed, disabled, or highlighted. The *lpNewItem* and *wIDNewItem* parameters are ignored.

MF_STRING	Specifies that the menu item is a character string; the *lpNewItem* parameter points to the string for the menu item.
MF_UNCHECKED	Does not place a checkmark next to the menu item. No checkmark is the default if neither MF_CHECKED nor MF_UNCHECKED is set. If the application has supplied checkmark bitmaps (see **SetMenuItemBitmaps**), setting this flag displays the "checkmark off" bitmap next to the menu item.

MoveTo

Syntax DWORD MoveTo(hDC, X, Y)
function MoveTo(DC: HDC; X, Y: Integer): Longint;

This function moves the current position to the point specified by the *X* and *Y* parameters.

Parameters

hDC	**HDC** Identifies the device context.
X	**int** Specifies the logical *x*-coordinate of the new position.
Y	**int** Specifies the logical *y*-coordinate of the new position.

Return value The return value specifies the *x*- and *y*-coordinates of the previous position. The *y*-coordinate is in the high-order word; the *x*-coordinate is in the low-order word.

Comments Although the **MoveTo** function has no output, it affects other output functions that use the current position.

MoveWindow

Syntax void MoveWindow(hWnd, X, Y, nWidth, nHeight, bRepaint)
procedure MoveWindow(Wnd: HWnd; X, Y, Width, Height: Integer; Repaint: Bool);

This function causes a WM_SIZE message to be sent to the given window. The *X*, *Y*, *nWidth*, and *nHeight* parameters give the new size of the window.

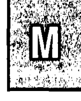
Parameters

hWnd	**HWND** Identifies a pop-up or child window.

X	**int** Specifies the new *x*-coordinate of the upper-left corner of the window.
Y	**int** Specifies the new *y*-coordinate of the upper-left corner of the window. For pop-up windows, *X* and *Y* are in screen coordinates (relative to the upper-left corner of the screen). For child windows, they are in client coordinates (relative to the upper-left corner of the parent window's client area).
nWidth	**int** Specifies the new width of the window.
nHeight	**int** Specifies the new height of the window.
bRepaint	**BOOL** Specifies whether or not the window is repainted after moving. If *bRepaint* is zero, the window is not repainted.

Return value None.

Comments Any child or pop-up window has a minimum width and height. These minimums depend on the style and content of the window. Any attempt to make the width and height smaller than the minimum by using the **MoveWindow** function will fail. The WM_SIZE message created by this function gives the new width and height of the client area of the window, not of the full window.



MulDiv 3.0

Syntax int MulDiv(nNumber, nNumerator, nDenominator)
function MulDiv(Number, Numerator, Denominator: Integer): Integer;

This function multiplies two word-length values and then divides the result by a third word-length value. The return value is the final result, rounded to the nearest integer.

Parameters

nNumber	**int** Specifies the number to be multiplied by *nNumerator*.
nNumerator	**int** Specifies the number to be multiplied by *nNumber*.
nDenominator	**int** Specifies the number by which the result of the multiplication is to be divided.

Return value The return value is the result of the multipliation and division. The return value is 32,767 or –32,767 if either an overflow occurred or *wDenominator* was zero.

NetBIOSCall 3.0

Syntax procedure NetBIOSCall;

This function allows an applications to issue the NETBIOS interrupt 5CH. An application should call this function instead of directly issuing a NETBIOS 5CH interrupt to preserve compatibility with future Microsoft products.

An application can call this function only from an assembly-language routine. It is exported from KERNEL.EXE and is not defined in any Windows include files.

To use this function call, an application should declare it in an assembly-language program as shown:

```
extrn    NETBIOSCALL    :far
```

If the application includes CMACROS.INC, the application declares it as shown:

```
externFP  NetBIOSCall
```

Before calling **NetBIOSCall**, all registers must be set as for an actual INT 5CH. All registers at the function's exit are the same as for the corresponding INT 5CH function.

Parameters None.

Return value None.

The following is an example of how to use the **NetBIOSCall** function:

```
extrn NETBIOSCALL : far
:
;set registers
cCall NetBIOSCall
```

OemKeyScan 3.0

Syntax DWORD OemKeyScan(wOemChar)
function OemKeyScan(OemChar: Word): Longint;

This function maps OEM ASCII codes 0 through 0x0FF into the OEM scan codes and shift states. It provides information which allows a program to send OEM text to another program by simulating keyboard input and is used specifically for this purpose by Windows in 386 enhanced mode.

Parameters *wOemChar* **WORD** Specifies the ASCII value of the OEM character.

Return value The return value contains in its low-order word the scan code of the OEM character identified by the *wOemChar* parameter. The high-order word of the return value contains flags which indicate the shift state. The following lists the flag bits and their meanings:

Bit	Meaning
2	CTRL key is pressed.
1	Either SHIFT key is pressed.

If the character is not defined in the OEM character tables, both the low-order and high-order words of the return value contain –1.

Comments This function does not provide translations for characters which require CTRL-ALT or dead keys. Characters not translated by this function must be copied by simulating input using the "ALT + keypad" mechanism. The NUMLOCK key must be off.

This function calls the **VkKeyScan** function in recent versions of the keyboard drivers.

OemToAnsi

Syntax int OemToAnsi(lpOemStr, lpAnsiStr)
function OemToAnsi(OemStr, AnsiStr: PChar): Bool;

This function translates the string pointed to by the *lpOemStr* parameter from the OEM-

defined character set into the ANSI character set. The string can be greater than 64K in length.

Parameters

lpOemStr **LPSTR** Points to a null-terminated string of characters from the OEM-defined character set.

lpAnsiStr **LPSTR** Points to the location where the translated string is to be copied. The *lpAnsiStr* parameter can be the same as *lpOemStr* to translate the string in place.

Return value The return value is always –1.

OemToAnsiBuff

Syntax void OemToAnsiBuff(lpOemStr, lpAnsiStr, nLength)
procedure OemToAnsiBuff(OemStr, AnsiStr: PChar; Length: Integer);

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

Parameters

lpOemStr — **LPSTR** Points to a buffer containing one or more characters from the OEM-defined character set.

lpAnsiStr — **LPSTR** Points to the location where the translated string is to be copied. The *lpAnsiStr* parameter can be the same as *lpOemStr* to translate the string in place.

nLength — **WORD** Specifies the number of characters in the buffer identified by the *lpOemStr* parameter. If *nLength* is zero, the length is 64K (65,536).

Return value None.

OffsetClipRgn

Syntax int OffsetClipRgn(hDC, X, Y)
function OffsetClipRgn(DC: HDC; X, Y: Integer): Integer;

This function moves the clipping region of the given device by the specified offsets. The function moves the region *X* units along the *x*-axis and *Y* units along the *y*-axis.

Parameters

hDC — **HDC** Identifies the device context.

X — **int** Specifies the number of logical units to move left or right.

Y — **int** Specifies the number of logical units to move up or down.

Return value The return value specifies the new region's type. It can be any one of the following values:

Value	Meaning
COMPLEXREGION	Clipping region has overlapping borders.
ERROR	Device context is not valid.
NULLREGION	Clipping region is empty.
SIMPLEREGION	Clipping region has no overlapping borders.

OffsetRect

Syntax

void OffsetRect(lpRect, X, Y)
procedure OffsetRect(var Rect: TRect; X, Y: Integer);

This function moves the given rectangle by the specified offsets. The **OffsetRect** function moves the rectangle *X* units along the *x*-axis and *Y* units along the *y*-axis. The *X* and *Y* parameters are signed values, so the rectangle can be moved left or right, and up or down.

Parameters

lpRect	**LPRECT** Points to a **RECT** data structure that contains the rectangle to be moved.
X	**int** Specifies the amount to move left or right. It must be negative to move left.
Y	**int** Specifies the amount to move up or down. It must be negative to move up.

Return value

None.

Comments

The coordinate values of a rectangle must not be greater than 32,767 or less than –32,768. The *X* and *Y* parameters must be chosen carefully to prevent invalid rectangles.

OffsetRgn

Syntax

int OffsetRgn(hRgn, X, Y)
function OffsetRgn(Rgn: HRgn; X, Y: Integer): Integer;

This function moves the given region by the specified offsets. The function moves the region *X* units along the *x*-axis and *Y* units along the *y*-axis.

Parameters

hRgn	**HRGN** Identifies the region to be moved.
X	**int** Specifies the number of units to move left or right.
Y	**int** Specifies the number of units to move up or down.

Return value

The return value specifies the new region's type. It can be any one of the following values:

Value	Meaning
COMPLEXREGION	Region has overlapping borders.
ERROR	Region handle is not valid.
NULLREGION	Region is empty.
SIMPLEREGION	Region has no overlapping borders.

Comments The coordinate values of a region must not be greater than 32,767 or less than –32,768. The *X* and *Y* parameters must be carefully chosen to prevent invalid regions.

OffsetViewportOrg

Syntax DWORD OffsetViewportOrg(hDC, X, Y)
function OffsetViewportOrg(DC: HDC; X, Y: Integer): Longint;

This function modifies the viewport origin relative to the current values. The formulas are written as follows:

xNewVO = xOldVO + X
yNewVO = yOldVO + Y

The new origin is the sum of the current origin and the *X* and *Y* values.

Parameters

hDC	**HDC** Identifies the device context.
X	**int** Specifies the number of device units to add to the current origin's *x*-coordinate.
Y	**int** Specifies the number of device units to add to the current origin's *y*-coordinate.

Return value The return value specifies the previous viewport origin (in device coordinates). The previous *y*-coordinate is in the high-order word; the previous *x*-coordinate is in the low-order word.

OffsetWindowOrg

Syntax DWORD OffsetWindowOrg(hDC, X, Y)
function OffsetWindowOrg(DC: HDC; X, Y: Integer): Longint;

zThis function modifies the viewport origin relative to the current values. The formulas are written as follows:

xNewWO = xOldWO + X
yNewWO = yOldWO + Y

The new origin is the sum of the current origin and the *X* and *Y* values.

Parameters

hDC	**HDC** Identifies the device context.
X	**int** Specifies the number of logical units to add to the current origin's *x*-coordinate.

	Y	**int** Specifies the number of logical units to add to the current origin's *y*-coordinate.

Return value The return value specifies the previous window origin (in logical coordinates). The previous *y*-coordinate is in the high-order word; the previous *x*-coordinate is in the low-order word.

OpenClipboard

Syntax BOOL OpenClipboard(hWnd)
function OpenClipboard(Wnd: HWnd): Bool;

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

Parameters	*hWnd*	**HWND** Identifies the window to be associated with the open clipboard.

Return value The return value specifies the status of the clipboard. It is nonzero if the clipboard is opened. If the clipboard has already been opened by another application, the return value is zero.

Comments An application should call the **CloseClipboard** function for every successful call to the **OpenClipboard** function.

OpenComm

Syntax int OpenComm(lpComName, wInQueue, wOutQueue)
function OpenComm(ComName: PChar; InQueue, OutQueue: Word): Integer;

This function opens a communication device and assigns an *nCid* handle to it. The communication device is initialized to a default configuration. The **SetCommState** function should be used to initialize the device to alternate values. The **OpenComm** function allocates space for receive and transmit queues. The queues are used by the interrupt-driven transmit/receive software.

Parameters	*lpComName*	**LPSTR** Points to a string which contains COM*n* or LPT*n*, where *n* ranges from 1 to the number of communication devices available for the particular type of I/O port.
	wInQueue	**WORD** Specifies the size of the receive queue.
	wOutQueue	**WORD** Specifies the size of the transmit queue.

Return value The return value specifies the open communication device. If an error occurs, the return value is one of the following negative error values:

Parameters

IE_BADID	Invalid or unsupported ID.
IE_BAUDRATE	Unsupported baud rate.
IE_BYTESIZE	Invalid byte size.
IE_DEFAULT	Error in default parameters.
IE_HARDWARE	Hardware not present.
IE_MEMORY	Unable to allocate queues.
IE_NOPEN	Device not open.
IE_OPEN	Device already open.

Comments LPT ports are not interrupt driven. For these ports, the *nInQueue* and *nOutQueue* parameters are ignored, and the queue size is set to zero.

OpenFile

Syntax int OpenFile(lpFileName, lpReOpenBuff, wStyle)
function OpenFile(FileName: PChar; var ReOpenBuff: TOFStruct; Style: Word): Integer;

This function creates, opens, reopens, or deletes a file.

Parameters

lpFileName **LPSTR** Points to a null-terminated character string that names the file to be opened. The string must consist of characters from the ANSI character set.

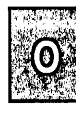
lpReOpenBuff **LPOFSTRUCT** Points to the **OFSTRUCT** data structure that is to receive information about the file when the file is first opened. The structure can be used in subsequent calls to the **OpenFile** function to refer to the open file.

The **szPathName** field of this data structure contains characters from the OEM character set.

wStyle **WORD** Specifies the action to take. These styles can be combined by using the bitwise OR operator:

Value	Meaning
OF_CANCEL	Adds a Cancel button to the OF_PROMPT dialog box. Pressing the Cancel button

	directs **OpenFile** to return a file-not-found error message.
OF_CREATE	Directs **OpenFile** to create a new file. If the file already exists, it is truncated to zero length.
OF_DELETE	Deletes the file.
OF_EXIST	Opens the file, and then closes it. Used to test for file existence.
OF_PARSE	Fills the **OFSTRUCT** data structure but carries out no other action.
OF_PROMPT	Displays a dialog box if the requested file does not exist. The dialog box informs the user that Windows cannot find the file and prompts the user to insert the file in drive A.
OF_READ	Opens the file for reading only.
OF_READWRITE	Opens the file for reading and writing.
OF_REOPEN	Opens the file using information in the re-open buffer.
OF_SHARE_COMPAT	Opens the file with compatibility mode, allowing any process on a given machine to open the file any number of times. **OpenFile** fails if the file has been opened with any of the other sharing modes.
OF_SHARE_DENY_NONE	Opens the file without denying other processes read or write access to the file. **OpenFile** fails if the file has been opened in compatibility mode by any other process.
OF_SHARE_DENY_READ	Opens the file and denies other processes read access to



	the file. **OpenFile** fails if the file has been opened in compatibility mode or for read access by any other process.
OF_SHARE_DENY_WRITE	Opens the file and denies other processes write access to the file. **OpenFile** fails if the file has been opened in compatibility or for write access by any other process.
OF_SHARE_EXCLUSIVE	Opens the file with exclusive mode, denying other processes both read and write access to the file. **OpenFile** fails if the file has been opened in any other mode for read or write access, even by the current process.
OF_VERIFY	Verifies that the date and time of the file are the same as when it was previously opened. Useful as an extra check for read-only files.
OF_WRITE	Opens the file for writing only.

Return value The return value specifies a DOS file handle if the function is successful. Otherwise, it is

–1.

Comments If the *lpFileName* parameter specifies a filename and extension only, this function searches for a matching file in the following directories:

1. The current directory.
2. The Windows directory (the directory containing WIN.COM); the **GetWindowsDirectory** function obtains the pathname of this directory.
3. The Windows system directory (the directory containing such system files as KERNEL.EXE); the **GetSystemDirectory** function obtains the pathname of this directory.
4. Any of the directories listed in the PATH environment variable.

5. Any directory in the list of directories mapped in a network.

Windows searches the directories in the listed order.

The *lpFileName* parameter cannot contain wildcard characters.

To close the file after use, the application should call the **_lclose** function.

OpenIcon

Syntax BOOL OpenIcon(hWnd)
function OpenIcon(Wnd: HWnd): Bool;

This function activates and displays an iconic (minimized) window. Windows restores it to its original size and position.

Parameters *hWnd* **HWND** Identifies the window.

Return value The return value specifies the outcome of the function. It is nonzero if the function is successful. Otherwise, it is zero.

OpenSound

Syntax int OpenSound( )
function OpenSound: Integer;

This function accesses the play device and prevents it from being opened subsequently by other applications.

Parameters None.

Return value The return value specifies the number of voices available. The return value is S_SERDVNA if the play device is in use, and S_SEROFM if insufficient memory is available.

OutputDebugString 3.0

Syntax void OutputDebugString(lpOutputString)
procedure OutputDebugString(OutputString: PChar);

This function sends a debugging message to the debugger if present, or to the auxiliary (AUX) device if the debugger is not present.

Parameters *lpOutputString* **LPSTR** Points to a null-terminated string.

Return value None.

Comments This function preserves all registers. It is available only in the debugging version of Windows.

PaintRgn

Syntax BOOL PaintRgn(hDC, hRgn)
function PaintRgn(DC: HDC; Rgn: HRgn): Bool;

This function fills the region specified by the *hRgn* parameter with the selected brush.

Parameters

hDC	**HDC** Identifies the device context that contains the region.
hRgn	**HRGN** Identifies the region to be filled. The coordinates for the given region are specified in device units.

Return value The return value specifies the outcome of the function. It is nonzero if the function is successful. Otherwise, it is zero.

PALETTEINDEX 3.0

Syntax COLORREF PALETTEINDEX(nPaletteIndex)
function PaletteIndex: Integer): TColorRef;

This macro accepts an index to a logical color palette entry and returns a value consisting of 1 in the high-order byte and the palette entry index in the low-order bytes. This is called a palette-entry specifier. An application using a color palette can pass this specifier instead of an explicit RGB value to functions that expect a color. This allows the function to use the color in the specified palette entry.

Parameters

nPaletteIndex	**int** Specifies an index to the palette entry containing the color to be used for a graphics operation.

Return value The return value is a logical-palette index specifier. When using a logical palette, an application can use this specifier in place of an explicit RGB value for GDI functions that require a color.

PALETTERGB 3.0

Syntax COLORREF PALETTERGB(cRed, cGreen, cBlue)

function PaletteRGB(R: Byte; G: Byte; B: Byte): Longint;

This macro accepts three values representing relative intensities of red, green, and blue, and returns a value consisting of 2 in the high-order byte and an RGB value in the three low-order bytes. This is called a palette-relative RGB specifier. An application using a color palette can pass this specifier instead of an explicit RGB value to functions that expect a color.

For output devices that support logical palettes, Windows matches a palette-relative RGB value to the nearest color in the logical palette of the device context, as though the application had specified an index to that palette entry. If an output device does not support a system palette, then Windows uses the palette-relative RGB as though it were a conventional RGB **DWORD** returned by the **RGB** macro.

Parameters

cRed	**BYTE** Specifies the intensity of the red color field.
cGreen	**BYTE** Specifies the intensity of the green color field.
cBlue	**BYTE** Specifies the intensity of the blue color field.

Return value The return value specifies a palette-relative RGB value.

PatBlt

Syntax BOOL PatBlt(hDC, X, Y, nWidth, nHeight, dwRop)
function PatBlt(DC: HDC; X, Y, Width, Height: Integer; Rop: Longint): Bool;

This function creates a bit pattern on the specified device. The pattern is a combination of the selected brush and the pattern already on the device. The raster-operation code specified by the *dwRop* parameter defines how the patterns are to be combined.

Parameters

hDC	**HDC** Identifies the device context.
X	**int** Specifies the logical *x*-coordinate of the upper-left corner of the rectangle that is to receive the pattern.
Y	**int** Specifies the logical *y*-coordinate of the upper-left corner of the rectangle that is to receive the pattern.
nWidth	**int** Specifies the width (in logical units) of the rectangle that is to receive the pattern.
nHeight	**int** Specifies the height (in logical units) of the rectangle that is to receive the pattern.

dwRop	**DWORD** Specifies the raster-operation code. Raster-operation codes (ROPs) define how GDI combines colors in output operations that involve a current brush, a possible source bitmap, and a destination bitmap. For a list of the raster-operation codes, see Table 4.12, "Raster Operations."

Return value The return value specifies the outcome of the function. It is nonzero if the bit pattern is drawn. Otherwise, it is zero.

Comments The values of *dwRop* for this function are a limited subset of the full 256 ternary raster-operation codes; in particular, an operation code that refers to a source cannot be used.

Not all devices support the **PatBlt** function. For more information, see the RC_BITBLT capability in the **GetDeviceCaps** function, earlier in this chapter.

Table 4.12 lists the various raster-operation codes for the *dwRop* parameter:

Table 4.12 Raster operations

Code	Description
PATCOPY	Copies pattern to destination bitmap.
PATINVERT	Combines destination bitmap with pattern using the Boolean OR operator.
DSTINVERT	Inverts the destination bitmap.
BLACKNESS	Turns all output black.
WHITENESS	Turns all output white.

PeekMessage

Syntax BOOL PeekMessage(lpMsg, hWnd, wMsgFilterMin, wMsgFilterMax, wRemoveMsg)
function PeekMessage(var Msg: TMsg; Wnd: HWnd; MsgFilterMin, MsgFilterMax, RemoveMsg: Word): Bool;

This function checks the application queue for a message and places the message (if any) in the data structure pointed to by the *lpMsg* parameter. Unlike the **GetMessage** function, the **PeekMessage** function does not wait for a message to be placed in the queue before returning. It does, however, yield control (if the PM_NOYIELD flag isn't set) and does not return control after the yield until Windows returns control to the application.

PeekMessage retrieves only messages associated with the window specified by the *hWnd* parameter, or any of its children as specified by the **IsChild** function, and within the range of message values given by the

wMsgFilterMin and *wMsgFilterMax* parameters. If *hWnd* is NULL, **PeekMessage** retrieves messages for any window that belongs to the application making the call. (The **PeekMessage** function does not retrieve messages for windows that belong to other applications.) If *hWnd* is –1, **PeekMessage** returns only messages with a *hWnd* of NULL as posted by the **PostAppMessage** function. If *wMsgFilterMin* and *wMsgFilterMax* are both zero, **PeekMessage** returns all available messages (no range filtering is performed).

The WM_KEYFIRST and WM_KEYLAST flags can be used as filter values to retrieve all key messages; the WM_MOUSEFIRST and WM_MOUSELAST flags can be used to retrieve all mouse messages.

Parameters

lpMsg — **LPMSG** Points to an **MSG** data structure that contains message information from the Windows application queue.

hWnd — **HWND** Identifies the window whose messages are to be examined.

wMsgFilterMin — **WORD** Specifies the value of the lowest message position to be examined.

wMsgFilterMax — **WORD** Specifies the value of the highest message position to be examined.

wRemoveMsg — **WORD** Specifies a combination of the flags described in the following list. PM_NOYIELD can be combined with either PM_NOREMOVE or PM_REMOVE:

Value	**Meaning**
PM_NOREMOVE	Messages are not removed from the queue after processing by **PeekMessage**.
PM_NOYIELD	Prevents the current task from halting and yielding system resources to another task.
PM_REMOVE	Messages are removed from the queue after processing by **PeekMessage**.

Return value

The return value specifies whether or not a message is found. It is nonzero if a message is available. Otherwise, it is zero.

Comments

PeekMessage does not remove WM_PAINT messages from the queue. The messages remain in the queue until processed. The **GetMessage**, **PeekMessage**, and **WaitMessage** functions yield control to other applications. These calls are the only way to let other applications run. If

your application does not call any of these functions for long periods of time, other applications cannot run.

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

If the application has stored long pointers to objects in the data or stack segment (global or local variables), and if they are unlocked, these pointers can become invalid after a call to **GetMessage, PeekMessage,** or **WaitMessage**. The *lpMsg* parameter of the called function remains valid in any case.

Pie

Syntax

BOOL Pie(hDC, X1, Y1, X2, Y2, X3, Y3, X4, Y4)
function Pie(DC: HDC; X1, Y1, X2, Y2, X3, Y3, X4, Y4: Integer): Bool;

This function draws a pie-shaped wedge by drawing an elliptical arc whose center and two endpoints are joined by lines. The center of the arc is the center of the bounding rectangle specified by the *X1*, *Y1*, *X2*, and *Y2* parameters. The starting and ending points of the arc are specified by the *X3*, *Y3*, *X4*, and *Y4* parameters. The arc is drawn with the selected pen, moving in a counterclockwise direction. Two additional lines are drawn from each endpoint to the arc's center. The pie-shaped area is filled with the selected brush.

If *X3* equals *X4* and *Y3* equals *Y4*, the result is an ellipse with a single line from the center of the ellipse to the point *(X3, Y3)*, or *(X4, Y4)*.

Parameters

hDC	**HDC** Identifies the device context.
X1	**int** Specifies the logical *x*-coordinate of the upper-left corner of the bounding rectangle.
Y1	**int** Specifies the logical *y*-coordinate of the upper-left corner of the bounding rectangle.
X2	**int** Specifies the logical *x*-coordinate of the lower-right corner of the bounding rectangle.
Y2	**int** Specifies the logical *y*-coordinate of the lower-right corner of the bounding rectangle.
X3	**int** Specifies the logical *x*-coordinate of the starting point of the arc. This point does not have to lie exactly on the arc.

Y3	**int** Specifies the logical *y*-coordinate of the starting point of the arc. This point does not have to lie exactly on the arc.
X4	**int** Specifies the logical *x*-coordinate of the endpoint of the arc. This point does not have to lie exactly on the arc.
Y4	**int** Specifies the logical *y*-coordinate of the endpoint of the arc. This point does not have to lie exactly on the arc.

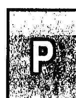
Return value The return value specifies whether or not the pie shape is drawn. It is nonzero if the pie shape is drawn. Otherwise, it is zero.

Comments The width of the rectangle, specified by the absolute value of *X2* – *X1*, must not exceed 32,767 units. This limit applies to the height of the rectangle as well. The current position is neither used nor updated by this function.

PlayMetaFile

Syntax BOOL PlayMetaFile(hDC, hMF)
function PlayMetaFile(DC: HDC; MF: THandle): Bool;

This function plays the contents of the specified metafile on the given device. The metafile can be played any number of times.



Parameters

hDC	**HDC** Identifies the device context of the output device.
hMF	**HANDLE** Identifies the metafile.

Return value The return value specifies the outcome of the function. It is nonzero if the function is successful. Otherwise, it is zero.

PlayMetaFileRecord

Syntax void PlayMetaFileRecord(hDC, lpHandletable, lpMetaRecord, nHandles)
procedure PlayMetaFileRecord(DC: HDC; var HandleTable: THandleTable; var MetaRecord: TMetaRecord; Handles: Word);

This function plays a metafile record by executing the GDI function call contained within the metafile record.

Parameters

hDC	**HDC** Identifies the device context of the output device.
lpHandletable	**LPHANDLETABLE** Points to the object handle table to be used for the metafile playback.
lpMetaRecord	**LPMETARECORD** Points to the metafile to be played.

nHandles	**WORD** Specifies the number of handles in the handle table.

Return value None.

Comments An application typically uses this function in conjunction with the **EnumMetafile** function to modify and then play a metafile.

Polygon

Syntax BOOL Polygon(hDC, lpPoints, nCount)
function Polygon(DC: HDC; var Points; Count: Integer): Bool;

This function draws a polygon consisting of two or more points (vertices) connected by lines. The polygons are filled using the current polygon-filling mode. For a description of the polygon-filling mode, see the **SetPolyFillMode** function, later in this chapter. The polygon is automatically closed, if necessary, by drawing a line from the last vertex to the first.

Parameters

hDC	**HDC** Identifies the device context.
lpPoints	**LPPOINT** Points to an array of points that specify the vertices of the polygon. Each point in the array is a **POINT** data structure.
nCount	**int** Specifies the number of vertices given in the array.

Return value The return value specifies the outcome of the function. It is nonzero if the function is successful. Otherwise, it is zero.

Comments The current position is neither used nor updated by this function.

The current polygon-filling mode can be retrieved or set by using the **GetPolyFillMode** and **SetPolyFillMode** functions.

Polyline

Syntax BOOL Polyline(hDC, lpPoints, nCount)
function Polyline(DC: HDC; var Points; Count: Integer): Bool;

This function draws a set of line segments, connecting the points specified by the *lpPoints* parameter. The lines are drawn from the first point through subsequent points with the result as if the **MoveTo** and **LineTo** functions were used to move to each new point and then connect it to the next.

However, the current position is neither used nor updated by the **Polyline** function.

Parameters

hDC	**HDC** Identifies the device context.
lpPoints	**LPPOINT** Points to an array of points to be connected. Each point in the array is a **POINT** data structure.
nCount	**int** Specifies the number of points in the array. The *nCount* parameter must be at least 2.

Return value The return value specifies whether or not the line segments were drawn. It is nonzero if the line segments were drawn. Otherwise, it is zero.

Comments This function draws lines with the selected pen.

PolyPolygon 3.0

Syntax BOOL PolyPolygon(hDC, lpPoints, lpPolyCounts, nCount)
function PolyPolygon(DC: HDC; var Points; var PolyCounts; Count: Integer): Bool;

This function creates a series of closed polygons. The polygons are filled using the current polygon-filling mode. For a description of the polygon-filling mode, see the **SetPolyFillMode** function, later in this chapter. The polygons may overlap, but they do not have to overlap.

Parameters

hDC	**HDC** Identifies the device context.
lpPoints	**LPPOINT** Points to an array of **POINT** data structures that define the vertices of the polygons. Each polygon must be a closed polygon. Unlike polygons created by the **Polygon** function, the polygons created by **PolyPolygon** are not automatically closed. The polygons are specified consecutively.
lpPolyCounts	**LPINT** Points to an array of integers, each of which specifies the number of points in one of the polygons in the *lpPoints* array.
nCount	**int** Specifies the total number of integers in the *lpPolyCounts* array.

Return value The return value specifies the outcome of the function. It is nonzero if the polygons were drawn. Otherwise, it is zero.

PostAppMessage

Syntax BOOL PostAppMessage(hTask, wMsg, wParam, lParam)
function PostAppMessage(Task: THandle; Msg, wParam: Word; lParam: Longint): Bool;

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

Parameters

hTask	**HANDLE** Identifies the task that is to receive the message. The **GetCurrentTask** function returns this handle.
wMsg	**WORD** Specifies the type of message posted.
wParam	**WORD** Specifies additional message information.
lParam	**DWORD** Specifies additional message information.

Return value The return value specifies whether or not the message is posted. It is nonzero if the message is posted. Otherwise, it is zero.

PostMessage

Syntax BOOL PostMessage(hWnd, wMsg, wParam, lParam)
function PostMessage(Wnd: HWnd; Msg, wParam: Word; lParam: Longint): Bool;

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

Parameters

hWnd	**HWND** Identifies the window to receive the message. If the *hWnd* parameter is 0xFFFF, the message is sent to all overlapped or pop-up windows in the system. The message is not sent to child windows.
wMsg	**WORD** Specifies the type of message posted.
wParam	**WORD** Specifies additional message information.
lParam	**DWORD** Specifies additional message information.

Return value The return value specifies whether or not the message is posted. It is nonzero if the message is posted. Otherwise, it is zero.

Comments An application should never use the **PostMessage** function to send a message to a control. If a system running Windows is configured for an expanded-memory system (EMS) and an application sends a message (by using the **PostMessage** function) with related data (that are pointed to by the *lParam* parameter) to a second application, the first application must place the data (that *lParam* points to) in global memory allocated with the **GlobalAlloc** function and the **GMEM_LOWER** flag. Note that this allocation of memory is necessary only if *lParam* contains a pointer.

Unlike other Windows functions, an application may call **PostMessage** at the hardwareinterrupt level.

PostQuitMessage

Syntax void PostQuitMessage(nExitCode)
procedure PostQuitMessage(ExitCode: Integer);

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

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

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

Parameters *nExitCode* **int** Specifies an application exit code. It is used as the *wParam* parameter of the WM_QUIT message.

Return value None.

ProfClear 3.0

Syntax void ProfClear( )

When running the Microsoft Windows Profiler, this function discards all samples currently in the sampling buffer. See *Tools* for more information on using the Profiler.

Parameters None.

Return value None.

ProfFinish 3.0

Syntax void ProfFinish( )

When running the Microsoft Windows Profiler, this function stops sampling and flushes the output buffer to disk.

When running with Windows in 386 enhanced mode, **ProfFinish** also frees the buffer for system use. See *Tools* for more information on using the Profiler.

Parameters None.

Return value None.

ProfFlush 3.0

Syntax void ProfFlush( )

When running the Microsoft Windows Profiler, this function flushes the sampling buffer to disk, provided that samples do not exceed predefined limits.

When running with Windows in any mode other than 386 enhanced mode, you must specify the size of the output buffer and the amount of samples to be written to disk.

When running with Windows in 386 enhanced mode, an application calls the **ProfSetup** function to specify the size of the output buffer and the amount of samples to be written to disk.

See *Tools* for more information on using the Profiler.

Parameters None.

Return value None.

Comments Do not call **ProfFlush** repeatedly because it can seriously impair the performance of the application. Additionally, do not call the function when DOS may be unstable, as in interrupt handling.

ProfInsChk 3.0

Syntax int ProfInsChk( )

This function determines if the Microsoft Windows Profiler is installed. See *Tools* for more information on using the Profiler.

Parameters None.

Return value The return value specifies whether Profiler is installed and the version installed. The return value is zero if Profiler is not installed, 1 if the Windows Profiler is installed for a mode other than 386 enhanced mode, and 2 if the Windows 386 enhanced mode Profiler is installed.

ProfSampRate 3.0

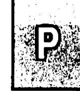
Syntax void ProfSampRate(nRate286, nRate386)

When running the Microsoft Windows Profiler, this function sets the rate of code sampling. See *Tools* for more information on using the Profiler.

Parameters

nRate286 **int** Specifies the sampling rate of Profiler if the application is running with Windows in any mode other than 386 enhanced mode. The value of *nRate286* ranges from 1 to 13, indicating the following sampling rates:

Value	Sampling Rate
1	122.070 microseconds
2	244.141 microseconds
3	488.281 microseconds
4	976.562 microseconds
5	1.953125 milliseconds
6	3.90625 milliseconds
7	7.8125 milliseconds
8	15.625 milliseconds
9	31.25 milliseconds
10	62.5 milliseconds
11	125 milliseconds
12	250 milliseconds
13	500 milliseconds



nRate386 **int** Specifies the sampling rate of Profiler if the application is running with Windows in 386 enhanced mode. The value of *nRate386* can range from 1 to 1000, specifying the sampling rate in milliseconds.

Return value None.

Comments The default rate is 5 (1.953125 milliseconds) for Windows in any mode other than 386 enhanced mode. The default rate is 2 milliseconds for Windows in 386 enhanced mode.

Profiler only selects the parameter appropriate for the version of Windows being used.

ProfSetup 3.0

Syntax void ProfSetup(nBufferSize, nSamples)

When running the Microsoft Windows Profiler with Windows in 386 enhanced mode, this function specifies the size of the output buffer and the amount of samples written to disk.

Profiler ignores the **ProfSetup** function when running with Windows in any mode other than 386 enhanced mode. See *Tools* for more information on using the Profiler.

Parameters

nBufferSize	**int** Specifies the size of the output buffer in kilobytes. The *nBufferSize* parameter can range from 1 to 1064. The default is 64.
nSamples	**int** Specifies how much sampling data Profiler writes to disk. A value of zero specifies unlimited sampling data. The default is zero.

ProfStart 3.0

Syntax void ProfStart( )

When running the Microsoft Windows Profiler, this function starts sampling. See *Tools* for more information on using the Profiler.

Parameters None.

Return value None.

ProfStop 3.0

Syntax void ProfStop( )

When running the Microsoft Windows Profiler, this function stops sampling. See *Tools* for more information on using the Profiler.

Parameters None.

Return value None.

PtInRect

Syntax BOOL PtInRect(lpRect, Point)
function PtInRect(var Rect: TRect; Point: TPoint): Bool;

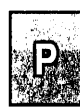
This function specifies whether the specified point lies within a given rectangle. A point is within a rectangle if it lies on the left or top side, or is within all four sides. A point on the right or bottom side is outside the rectangle.

Parameters

lpRect	**LPRECT** Points to a **RECT** data structure that contains the specified rectangle.
Point	**POINT** Specifies a **POINT** data structure that contains the specified point.

Return value The return value specifies whether the specified point lies within the given rectangle. It is nonzero if the point lies within the given rectangle. Otherwise, it is zero.

PtInRegion



Syntax BOOL PtInRegion(hRgn, X, Y)
function PtInRegion(Rgn: HRgn; X, Y: Integer): Bool;

This function specifies whether the point given by the *X* and *Y* parameters is in the given region.

Parameters

hRgn	**HRGN** Identifies the region to be examined.
X	**int** Specifies the logical *x*-coordinate of the point.
Y	**int** Specifies the logical *y*-coordinate of the point.

Return value The return value specifies whether the specified point is in the given region. It is nonzero if the point is in the region. Otherwise, it is zero.

PtVisible

Syntax BOOL PtVisible(hDC, X, Y)
function PtVisible(DC: HDC; X1, Y1, X2, Y2; Integer): Bool;

This function specifies whether the given point is within the clipping region of the specified device context.

Parameters

hDC	**HDC** Identifies the device context.
X	**int** Specifies the logical *x*-coordinate of the point.
Y	**int** Specifies the logical *y*-coordinate of the point.

Return value The return value specifies whether the specified point is within the clipping region of the given display context. It is nonzero if the point is within the clipping region. Otherwise, it is zero.

ReadComm

Syntax

int ReadComm(nCid, lpBuf, nSize)
function ReadComm(Cid: Integer; Buf: PChar, Size: Integer): Integer;

This function reads the number of characters specified by the *nSize* parameter from the communication device specified by the *nCid* parameter and copies the characters into the buffer pointed to by the *lpBuf* parameter.

Parameters

nCid	**int** Specifies the communication device to be read. The **OpenComm** function returns this value.
lpBuf	**LPSTR** Points to the buffer that is to receive the characters read.
nSize	**int** Specifies the number of characters to be read.

Return value The return value specifies the number of characters actually read. It is less than the number specified by *nSize* only if the number of characters in the receive queue is less than that specified by *nSize*. If it is equal to *nSize*, additional characters may be queued for the device. If the return value is zero, no characters are present.

When an error occurs, the return value is set to a value less than zero, with the absolute value being the actual number of characters read. The cause of the error can be determined by using the **GetCommError** function to retrieve the error code and status. Since errors can occur when no bytes are present, if the return value is zero, the **GetCommError** function should be used to ensure that no error occurred.

For parallel I/O ports, the return value will always be zero.

RealizePalette 3.0

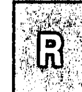
Syntax int RealizePalette(hDC)
function RealizePalette(DC: HDC): Word;

This function maps to the system palette entries in the logical palette currently selected into a device context.

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

Parameters *hDC* **HDC** Identifies the device context.

Return value The return value specifies how many entries in the logical palette were mapped to different entries in the system palette. This represents the number of entries which this function remapped to accommodate changes in the system palette since the logical palette was last realized.



Rectangle

Syntax BOOL Rectangle(hDC, X1, Y1, X2, Y2)
function Rectangle(DC: HDC; X1, Y1, X2, Y2: Integer): Bool;

This function draws a rectangle. The interior of the rectangle is filled by using the selected brush, and a border is drawn with the selected pen.

Parameters

hDC	**HDC** Identifies the device context.
X1	**int** Specifies the logical *x*-coordinate of the upper-left corner of the rectangle.
Y1	**int** Specifies the logical *y*-coordinate of the upper-left corner of the rectangle.

X2	**int** Specifies the logical *x*-coordinate of the lower-right corner of the rectangle.
Y2	**int** Specifies the logical *y*-coordinate of the lower-right corner of the rectangle.

Return value The return value specifies whether the rectangle is drawn. It is nonzero if the rectangle is drawn. Otherwise, it is zero.

Comments The width of the rectangle specified by the *X1*, *Y1*, *X2*, and *Y2* parameters must not exceed 32,767 units. This limit applies to the height of the rectangle as well.

The current position is neither used nor updated by this function.

RectInRegion 3.0

Syntax BOOL RectInRegion(hRegion, lpRect)
function RectInRegion(Rgn: HRgn; var Rect: TRect): Bool;

This function determines whether any part of the rectangle specified by the *lpRect* parameter is within the boundaries of the region identified by the *hRegion* parameter.

Parameters *hRegion*, **HRGN** Identifies the region.

lpRect, **LPRECT** Identifies the rectangle.

Return value The return value is TRUE if any part of the specified rectangle lies within the boundaries of the region. Otherwise, the return value is FALSE.

RectVisible

Syntax BOOL RectVisible(hDC, lpRect)
function RectVisible(DC: HDC; var Rect: TRect): Bool;

This function determines whether any part of the given rectangle lies within the clipping region of the specified display context.

Parameters

hDC	**HDC** Identifies the device context.
lpRect	**LPRECT** Points to a **RECT** data structure that contains the logical coordinates of the specified rectangle.

Return value The return value specifies whether the rectangle is within the clipping region. It is nonzero if some portion of the given rectangle lies within the clipping region. Otherwise, it is zero.

RegisterClass

Syntax BOOLRegisterClass(lpWndClass)
function RegisterClass(var WndClass: TWndClass): Bool;

This function registers a window class for subsequent use in calls to the **CreateWindow** function. The window class has the attributes defined by the contents of the data structure pointed to by the *lpWndClass* parameter. If two classes with the same name are registered, the second attempt fails and the information for that class is ignored.

Parameters

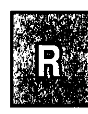
lpWndClass	**LPWNDCLASS** Points to a **WNDCLASS** data structure. The structure must be filled with the appropriate class attributes before being passed to the function. See the following "Comments" section for details.

Return value The return value specifies whether the window class is registered. It is nonzero if the class is registered. Otherwise, it is zero.

Comments The callback function must use the Pascal calling conventions and must be declared **FAR**.

Callback function

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



WndProc is a placeholder for the application-supplied function name. The actual name must be exported by including it in an **EXPORTS** statement in the application's module-definition file.

Parameters

hWnd	Identifies the window that receives the message.
wMsg	Specifies the message number.
wParam	Specifies additional message-dependent information.
lParam	Specifies additional message-dependent information.

Return value The window function returns the result of the message processing. The possible return values depend on the actual message sent.

RegisterClipboardFormat

Syntax WORD RegisterClipboardFormat(lpFormatName)
function RegisterClipboardFormat(FormatName: PChar): Word;

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

Parameters *lpFormatName* **LPSTR** Points to a character string that names the new format. The string must be a null-terminated character string.

Return value The return value specifies the newly registered format. If the identical format name has been registered before, even by a different application, the format's reference count is increased and the same value is returned as when the format was originally registered. The return value is zero if the format cannot be registered.

Comments The format value returned by the **RegisterClipboardFormat** function is within the range of 0xC000 to 0xFFFF.

RegisterWindowMessage

Syntax WORD RegisterWindowMessage(lpString)
function RegisterWindowMessage(Str: PChar): Word;

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

RegisterWindowMessage is typically used for communication between two cooperating applications.

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

Parameters *lpString* **LPSTR** Points to the message string to be registered.

Return value The return value specifies the outcome of the function. It is an unsigned short integer within the range 0xC000 to 0xFFFF if the message is successfully registered. Otherwise, it is zero.

Comments Use the **RegisterWindowMessage** function only when the same message must be understood by more than one application. For sending private messages within an application, an application can use any integer within the range WM_USER to 0xBFFF.

ReleaseCapture

Syntax void ReleaseCapture( )
procedure ReleaseCapture;

This function releases the mouse capture and restores normal input processing. A window with the mouse capture receives all mouse input regardless of the position of the cursor.

Parameters None.

Return value None.

Comments An application calls this function after calling the **SetCapture** function.

ReleaseDC

Syntax int ReleaseDC(hWnd, hDC)
function ReleaseDC(Wnd: HWnd; DC: HDC): Integer;

This function releases a device context, freeing it for use by other applications. The effect of the **ReleaseDC** function depends on the device-context type. It only frees common and window device contexts. It has no effect on class or private device contexts.

Parameters

hWnd	**HWND** Identifies the window whose device context is to be released.
hDC	**HDC** Identifies the device context to be released.

Return value The return value specifies whether the device context is released. It is 1 if the device context is released. Otherwise, it is zero.

Comments The application must call the **ReleaseDC** function for each call to the **GetWindowDC** function and for each call to the **GetDC** function that retrieves a common device context.

RemoveFontResource

Syntax BOOL RemoveFontResource(lpFilename)
function RemoveFontResourc(FileName: PChar): Bool;

This function removes an added font resource from the file named by the *lpFilename* parameter or from the Windows font table.

Parameters

lpFilename — **LPSTR** Points to a string that names the font-resource file or contains a handle to a loaded module. If *lpFilename* points to the font-resource filename, the string must be null-terminated and have the DOS filename format. If *lpFilename* contains a handle, the handle must be in the low-order word; the high-order word must be zero.

Return value The return value specifies the outcome of the function. It is nonzero if the function is successful. Otherwise, it is zero.

Comments Any application that adds or removes fonts from the Windows font table should notify other windows of the change by using the **SendMessage** function with the *hWnd* parameter set to –1 to send a WM_FONTCHANGE message to all top-level windows in the system. The **RemoveFontResource** function may not actually remove the font resource. If there are outstanding references to the resource, the font resource remains loaded until the last referencing logical font has been deleted by using the **DeleteObject** function.

RemoveMenu 3.0

Syntax BOOL RemoveMenu(hMenu, nPosition, wFlags)
function RemoveMenu(Menu: HMenu; Position, Flags: Word): Bool;

This function deletes an menu item with an associated pop-up menu from the menu identified by the *hMenu* parameter but does not destroy the handle for the pop-up menu, allowing the menu to be reused. Before calling this function, the application should call **GetSubMenu** to retrieve the pop-up menu handle.

Parameters

hMenu — **HMENU** Identifies the menu to be changed.

nPosition — **WORD** Specifies the menu item to be removed. The interpretation of the *nPosition* parameter depends upon the setting of the *wFlags* parameter.

If wFlags is:	nPosition
MF_BYCOMMAND	Specifies the command ID of the existing menu item.
MF_BYPOSITION	Specifies the position of the menu item. The first item in the menu is at position zero.

wFlags **WORD** Specifies how the *nPosition* parameter is interpreted. It must be either MF_BYCOMMAND or MF_BYPOSITION.

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

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

RemoveProp

Syntax HANDLE RemoveProp(hWnd, lpString)
function RemoveProp(Wnd: HWnd; Str: PChar): THandle;

This function removes an entry from the property list of the specified window. The character string specified by the *lpString* parameter identifies the entry to be removed.

The **RemoveProp** function returns the data handle associated with the string so that the application can free the data associated with the handle.

Parameters

hWnd **HWND** Identifies the window whose property list is to be changed.

lpString **LPSTR** Points to a null-terminated character string or to an atom that identifies a string. If an atom is given, it must have been previously created by means of the **AddAtom** function. The atom, a 16-bit value, must be placed in the low-order word of *lpString*; the high-order word must be zero.

Return value The return value identifies the given string. It is NULL if the string cannot be found in the given property list.

Comments An application must free the data handles associated with entries removed from a property list. The application should only remove those properties which it added to the property list.

ReplyMessage

Syntax void ReplyMessage(lReply)
procedure ReplyMessage(Reply: Longint);

This function is used to reply to a message sent through the **SendMessage** function without returning control to the function that called **SendMessage.**

By calling this function, the window function that receives the message allows the task that called **SendMessage** to continue to execute as though the task that received the message had returned control. The task that calls **ReplyMessage** also continues to execute.

Normally a task that calls **SendMessage** to send a message to another task will not continue executing until the window procedure that Windows calls to receive the message returns.

However, if a task that is called to receive a message needs to perform some type of operation that might yield control (such as calling the **MessageBox** or **DialogBox** functions), Windows could be placed in a deadlock situation where the sending task needs to execute and process messages but cannot because it is waiting for **SendMessage** to return.

An application can avoid this problem if the task receiving the message calls **ReplyMessage** before performing any operation that could cause the task to yield.

The **ReplyMessage** function has no effect if the message was not sent through the **SendMessage** function or if the message was sent by the same task.

Parameters *lReply* — **LONG** Specifies the result of the message processing. The possible values depend on the actual message sent.

Return value None.

ResizePalette 3.0

Syntax BOOL ResizePalette(hPalette, nNumEntries)
function ResizePalette(Palette: HPalette; NumEntries: Word): Bool;

This function changes the size of the logical palette specified by the *hPalette* parameter to the number of entries specified by the **nNumEntries** parameter. If an application calls **ResizePalette** to reduce the size of the palette, the entries remaining in the resized palette are unchanged.

If the application calls **ResizePalette** to enlarge the palette, the additional palette entries are set to black (the red, green, and blue values are all 0) and the flags for all additional entries are set to 0.

Parameters

hPalette	**HPALETTE** Identifies the palette to be changed.
nNumEntries	**int** Specifies the number of entries in the palette after it has been resized.

Return value The return value specifies the outcome of the function. It is TRUE if the palette was successfully resized. Otherwise, it is FALSE.

RestoreDC

Syntax BOOLRestoreDC(hDC, nSavedDC)
function RestoreDC(DC: HDC; SavedDC: Integer): Bool;

This function restores the device context specified by the *hDC* parameter to the previous state identified by the *nSavedDC* parameter.

The **RestoreDC** function restores the device context by copying state information saved on the context stack by earlier calls to the **SaveDC** function.

The context stack can contain the state information for several device contexts. If the context specified by *nSavedDC* is not at the top of the stack, **RestoreDC** deletes any state information between the device context specified by the *nSavedDC* parameter and the top of the stack. The deleted information is lost.

Parameters

hDC	**HDC** Identifies the device context.
nSavedDC	**int** Specifies the device context to be restored. It can be a value returned by a previous **SaveDC** function call. If *nSavedDC* is –1, the most recent device context saved is restored.

Return value The return value specifies the outcome of the function. It is TRUE if the specified context was restored. Otherwise, it is FALSE.

RGB

Syntax COLORREF RGB(cRed, cGreen, cBlue)
function RGB(R: Byte; G: Byte; B: Byte): Longint;

This macro selects an RGB color based on the parameters supplied and the color capabilities of the output device.

Parameters

cRed	**BYTE** Specifies the intensity of the red color field.
cGreen	**BYTE** Specifies the intensity of the green color field.
cBlue	**BYTE** Specifies the intensity of the blue color field.

Return value The return value specifies the resultant RGB color.

Comments The intensity for each argument can range from 0 to 255. If all three intensities are specified as 0, the result is black. If all three intensities are specified as 255, the result is white.

For more information on using color values in a color palette, see the descriptions of the **PALETTEINDEX** and **PALETTERGB** macros, earlier in this chapter.

RoundRect

Syntax BOOL RoundRect(hDC, X1, Y1, X2, Y2, X3, Y3)
function RoundRect(DC: HDC; X1, Y1, X2, Y2, X3, Y3: Integer): Bool;

This function draws a rectangle with rounded corners. The interior of the rectangle is filled by using the selected brush, and a border is drawn with the selected pen.

Parameters	*hDC*	**HDC** Identifies the device context.
	X1	**int** Specifies the logical *x*-coordinate of the upper-left corner of the rectangle.
	Y1	**int** Specifies the logical *y*-coordinate of the upper-left corner of the rectangle.
	X2	**int** Specifies the logical *x*-coordinate of the lower-right corner of the rectangle.
	Y2	**int** Specifies the logical *y*-coordinate of the lower-right corner of the rectangle.
	X3	**int** Specifies the width of the ellipse used to draw the rounded corners.
	Y3	**int** Specifies the height of the ellipse used to draw the rounded corners.

Return value The return value specifies whether the rectangle is drawn. It is nonzero if the rectangle is drawn. Otherwise, it is zero.

Comments The width of the rectangle specified by the *X1*, *Y1*, *X2*, and *Y2* parameters must not exceed 32,767 units. This limit applies to the height of the rectangle as well. The current position is neither used nor updated by this function.

SaveDC

Syntax int SaveDC(hDC)
function SaveDC(DC: HDC): Integer;

This function saves the current state of the device context specified by the *hDC* parameter by copying state information (such as clipping region, selected objects, and mapping mode) to a context stack. The saved device context can later be restored by using the **RestoreDC** function.

Parameters *hDC* **HDC** Identifies the device context to be saved.

Return value The return value specifies the saved device context. It is zero if an error occurs.

Comments The **SaveDC** function can be used any number of times to save any number of device-context states.

ScaleViewportExt

Syntax DWORD ScaleViewportExt(hDC, Xnum, Xdenom, Ynum, Ydenom)
function ScaleViewportExt(DC: HDC; Xnum, Xdenom, Ynum, Ydenom: Integer): Longint;

This function modifies the viewport extents relative to the current values. The formulas are written as follows:

xNewVE = (xOldVE × Xnum)/ X denom
yNewVE = (yOldVE × Ynum) / Ydenom

The new extent is calculated by multiplying the current extents by the given numerator and then dividing by the given denominator.

Parameters

hDC	**HDC** Identifies the device context.
Xnum	**int** Specifies the amount by which to multiply the current *x*-extent.
Xdenom	**int** Specifies the amount by which to divide the current *x*-extent.
Ynum	**int** Specifies the amount by which to multiply the current *y*-extent.
Ydenom	**int** Specifies the amount by which to divide the current *y*-extent.

Return value The return value specifies the previous viewport extents (in device units). The previous *y*-extent is in the high-order word; the previous *x*-extent is in the low-order word.

ScaleWindowExt

Syntax DWORD ScaleWindowExt(hDC, Xnum, Xdenom, Ynum, Ydenom)
function ScaleWindowExt(DC: HDC; Xnum, Xdenom, Ynum, Ydenom: Integer): Longint;

This function modifies the window extents relative to the current values. The formulas are written as follows:

xNewWE = (xOldWE × Xnum) / Xdenom
yNewWE = (yOldWE × Ynum) / Ydenom

The new extent is calculated by multiplying the current extents by the given numerator and then dividing by the given denominator.

Parameters

hDC	**HDC** Identifies the device context.
Xnum	**int** Specifies the amount by which to multiply the current *x*-extent.
Xdenom	**int** Specifies the amount by which to divide the current *x*-extent.
Ynum	**int** Specifies the amount by which to multiply the current *y*-extent.
Ydenom	**int** Specifies the amount by which to divide the current *y*-extent.

Return value The return value specifies the previous window extents (in logical units). The previous *y*-extent is in the high-order word; the previous *x*-extent is in the low-order word.

ScreenToClient

Syntax void ScreenToClient(hWnd, lpPoint)
procedure ScreenToClient(Wnd: HWnd; var Point: TPoint);

This function converts the screen coordinates of a given point on the display to client coordinates. The **ScreenToClient** function uses the window given by the *hWnd* parameter and the screen coordinates given in the **POINT** data structure pointed to by the *lpPoint* parameter to compute

client coordinates, and then replaces the screen coordinates with the client coordinates. The new coordinates are relative to the upper-left corner of the given window's client area.

Parameters

hWnd	**HWND** Identifies the window whose client area will be used for the conversion.
lpPoint	**LPPOINT** Points to a **POINT** data structure that contains the screen coordinates to be converted.

Return value None.

Comments The **ScreenToClient** formula assumes the given point is in screen coordinates.

ScrollDC

Syntax BOOL ScrollDC(hDC, dx, dy, lprcScroll, lprcClip, hrgnUpdate, lprcUpdate)
function ScrollDC(DC: HDC; dx, dy: Integer; var Scroll, Clip: TRect; UpdateRgn: HRgn; UpdateRect: PRect): Bool;

This function scrolls a rectangle of bits horizontally and vertically. The *lprcScroll* parameter points to the rectangle to be scrolled, the *dx* parameter specifies the number of units to be scrolled horizontally, and the *dy* parameter specifies the number of units to be scrolled vertically.

Parameters

hDC	**HDC** Identifies the device context that contains the bits to be scrolled.
dx	**int** Specifies the number of horizontal scroll units.
dy	**int** Specifies the number of vertical scroll units.
lprcScroll	**LPRECT** Points to the **RECT** data structure that contains the coordinates of the scrolling rectangle.
lprcClip	**LPRECT** Points to the **RECT** data structure that contains the coordinates of the clipping rectangle. When this rectangle is smaller than the original pointed to by *lprcScroll*, scrolling occurs only in the smaller rectangle.
hrgnUpdate	**HRGN** Identifies the region uncovered by the scrolling process. The **ScrollDC** function defines this region; it is not necessarily a rectangle.
lprcUpdate	**LPRECT** Points to the **RECT** data structure that, upon return, contains the coordinates of the rectangle that bounds the

scrolling update region. This is the largest rectangular area that requires repainting.

Return value This value specifies the outcome of the function. It is nonzero if scrolling is executed. Otherwise, it is zero.

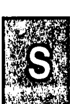
Comments If the *lprcUpdate* parameter is NULL, Windows does not compute the update rectangle. If both the *hrgnUpdate* and *lprcUpdate* parameters are NULL, Windows does not compute the update region. If *hrgnUpdate* is not NULL, Windows assumes that it contains a valid region handle to the region uncovered by the scrolling process (defined by the **ScrollDC** function).

An application should use the **ScrollWindow** function when it is necessary to scroll the entire client area of a window. Otherwise, it should use **ScrollDC**.

ScrollWindow

Syntax void ScrollWindow(hWnd, XAmount, YAmount, lpRect, lpClipRect)
procedure ScrollWindow(Wnd: HWnd; XAmount, YAmount: Integer; Rect, ClipRect: PRect);

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



Parameters

hWnd	**HWND** Identifies the window whose client area is to be scrolled.
XAmount	**int** Specifies the amount (in device units) to scroll in the *x* direction.
YAmount	**int** Specifies the amount (in device units) to scroll in the *y* direction.
lpRect	**LPRECT** Points to a **RECT** data structure that specifies the portion of the client area to be scrolled. If *lpRect* is NULL, the entire client area is scrolled.
lpClipRect	**LPRECT** Points to a **RECT** data structure that specifies the clipping rectangle to be scrolled. Only bits inside this

rectangle are scrolled. If *lpClipRect* is NULL, the entire window is scrolled.

Return value None.

Comments If the caret is in the window being scrolled, **ScrollWindow** automatically hides the caret to prevent it from being erased, then restores the caret after the scroll is finished. The caret position is adjusted accordingly.

The area uncovered by the **ScrollWindow** function is not repainted, but is combined into the window's update region. The application will eventually receive a WM_PAINT message notifying it that the region needs repainting. To repaint the uncovered area at the same time the scrolling is done, call the **UpdateWindow** function immediately after calling **ScrollWindow**.

If the *lpRect* parameter is NULL, the positions of any child windows in the window are offset by the amount specified by *XAmount* and *YAmount*, and any invalid (unpainted) areas in the window are also offset. **ScrollWindow** is faster when *lpRect* is NULL.

If the *lpRect* parameter is not NULL, the positions of child windows are *not* changed, and invalid areas in the window are *not* offset. To prevent updating problems when *lpRect* is not NULL, call the **UpdateWindow** function to repaint the window before calling **ScrollWindow**.

SelectClipRgn

Syntax int SelectClipRgn(hDC, hRgn)
function SelectClipRgn(DC: HDC; Rgn: HRgn): Integer;

This function selects the given region as the current clipping region for the specified device context. Only a copy of the selected region is used. The region itself can be selected for any number of other device contexts, or it can be deleted.

Parameters

hDC	**HDC** Identifies the device context.
hRgn	**HRGN** Identifies the region to be selected.

Return value The return value specifies the region's type. It can be any one of the following values:

Value	Meaning
COMPLEXREGION	New clipping region has overlapping borders.
ERROR	Device context or region handle is not valid.
NULLREGION	New clipping region is empty.
SIMPLEREGION	New clipping region has no overlapping borders.

Comments The **SelectClipRgn** function assumes that the coordinates for the given region are specified in device units.

Some printer devices support graphics at lower resolutions than text output to increase speed, but at the expense of quality. These devices scale coordinates for graphics so that one graphics device point corresponds to two or four true device points. This scaling factor affects clipping. If a region will be used to clip graphics, its coordinates must be divided down by the scaling factor. If the region will be used to clip text, no scaling adjustment is needed. The scaling factor is determined by using the **GETSCALINGFACTOR** printer escape.

SelectObject

Syntax HANDLE SelectObject(hDC, hObject)
function SelectObject(DC: HDC; hObject: THandle): THandle;

This function selects the logical object specified by the *hObject* parameter as the selected object of the specified device context. The new object replaces the previous object of the same type. For example, if *hObject* is the handle to a logical pen, the **SelectObject** function replaces the selected pen with the pen specified by *hObject*.

Selected objects are the default objects used by the GDI output functions to draw lines, fill interiors, write text, and clip output to specific areas of the device surface. Although a device context can have six selected objects (pen, brush, font, bitmap, region, and logical palette), no more than one object of any given type can be selected at one time. **SelectObject** does not select a logical palette; to select a logical palette, the application must use **SelectPalette**.

Parameters

hDC **HDC** Identifies the device context.

hObject **HANDLE** Identifies the object to be selected. It may be any one of the following, and must have been created by using one of the following functions:

Object	Function
Bitmap[1]	**CreateBitmap**
	CreateBitmapIndirect
	CreateCompatibleBitmap
	CreateDIBitmap
Brush	**CreateBrushIndirect**
	CreateHatchBrush
	CreatePatternBrush
	CreateSolidBrush
Font	**CreateFont**
	CreateFontIndirect
Pen	**CreatePen**
	CreatePenIndirect
Region	**CombineRgn**
	CreateEllipticRgn
	CreateEllipticRgnIndirect
	CreatePolygonRgn
	CreateRectRgn
	CreateRectRgnIndirect

[1] (Bitmaps can be selected for memory device contexts only, and for only one device context at a time.)

Return value The return value identifies the object being replaced by the object specified by the *hObject* parameter. It is NULL if there is an error.

If the *hDC* parameter specifies a metafile, the return value is nonzero if the function is successful. Otherwise, it is zero.

If a region is being selected, the return is the same as for **SelectClipRgn**.

Comments When you select a font, pen, or brush by using the **SelectObject** function, GDI allocates space for that object in its data segment. Because data-segment space is limited, you should use the **DeleteObject** function to delete each drawing object that you no longer need.

Before deleting the last of the unneeded drawing objects, an application should select the original (default) object back into the device context, unless the device context is a metafile. The **SelectObject** function does not return the previously selected object when the *hDC* parameter identifies a metafile device context. Calling **SelectObject** with the *hObject* parameter set to a value returned by a previous call to **SelectObject** can cause unpredictable results. Metafiles perform their own object cleanup. As a result, an application does not have to reselect default objects when recording a metafile.

An application cannot select a bitmap into more than one device context at any time.

SelectPalette 3.0

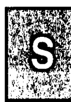
Syntax HPALETTE SelectPalette(hDC, hPalette, bForceBackground)
function SelectPalette(DC: HDC; Palette: HPalette; ForceBackground: Bool): HPalette;

This function selects the logical palette specified by the *hPalette* parameter as the selected palette object of the device context identified by the *hDC* parameter. The new palette becomes the palette object used by GDI to control colors displayed in the device context and replaces the previous palette.

Parameters

hDC	**HDC** Identifies the device context.
hPalette	**HPALETTE** Identifies the logical palette to be selected. **CreatePalette** creates a logical palette.
bForceBackground	**BOOL** Specifies whether the logical palette is forced to be a background palette. If *bForceBackground* is nonzero, the selected palette is always a background palette, regardless of whether the window has input focus. If *bForceBackground* is zero, the logical palette is a foreground palette when the window has input focus.

Return value The return value identifies the logical palette being replaced by the palette specified by the *hPalette* parameter. It is NULL if there is an error.



Comments An application can select a logical palette into more than one device context. However, changes to a logical palette will affect all device contexts for which it is selected. If an application selects a palette object into more than one device context, the device contexts must all belong to the same physical device (such as a display or printer).

SendDlgItemMessage

Syntax DWORD SendDlgItemMessage(hDlg, nIDDlgItem, wMsg, wParam, lParam)
function SendDlgItemMessage(Dlg: HWnd; IDDlgItem: Integer; Msg, wParam: Word; lParam: Longint): Longint;

This function sends a message to the control specified by the *nIDDlgItem* parameter within the dialog box specified by the *hDlg* parameter. The

SendDlgItemMessage function does not return until the message has been processed.

Parameters

hDlg	**HWND** Identifies the dialog box that contains the control.
nIDDlgItem	**int** Specifies the integer identifier of the dialog item that is to receive the message.
wMsg	**WORD** Specifies the message value.
wParam	**WORD** Specifies additional message information.
lParam	**DWORD** Specifies additional message information.

Return value The return value specifies the outcome of the function. It is the value returned by the control's window function, or zero if the control identifier is not valid.

Comments Using **SendDlgItemMessage** is identical to obtaining a handle to the given control and calling the **SendMessage** function.

SendMessage

Syntax DWORD SendMessage(hWnd, wMsg, wParam, lParam)
function SendMessage(Wnd: HWnd; Msg, wParam: Word; lParam: Longint): Longint;

This function sends a message to a window or windows. The **SendMessage** function does not return until the message has been processed. If the window that receives the message is part of the same application, the window function is called immediately as a subroutine. If the window is part of another task, Windows switches to the appropriate task and calls the appropriate window function, and then passes the message to the window function. The message is not placed in the destination application's queue.

Parameters

hWnd	**HWND** Identifies the window that is to receive the message. If the *hWnd* parameter is 0xFFFF, the message is sent to all pop-up windows in the system. The message is not sent to child windows.
wMsg	**WORD** Specifies the message to be sent.
wParam	**WORD** Specifies additional message information.
lParam	**DWORD** Specifies additional message information.

Return value The return value specifies the outcome of the function. It is the value returned by the window function that received the message; its value depends on the message being sent.

Comments If a system running Windows is configured for expanded memory (EMS) and an application sends a message (by using the **SendMessage** function) with related data (that is pointed to by the *lParam* parameter) to a second application, the first application must place the data (that *lParam* points to) in global memory allocated by the **GlobalAlloc** function and the GMEM_LOWER flag. Note that this allocation of memory is only necessary if *lParam* contains a pointer.

SetActiveWindow

Syntax HWND SetActiveWindow(hWnd)
function SetActiveWindow(Wnd: HWnd): HWnd;

This function makes a top-level window the active window.

Parameters

hWnd	**HWND** Identifies the top-level window to be activated.

Return value The return value identifies the window that was previously active. The **SetActiveWindow** function should be used with care since it allows an application to arbitrarily take over the active window and input focus. Normally, Windows takes care of all activation.

SetBitmapBits

Syntax LONG SetBitmapBits(hBitmap, dwCount, lpBits)
function SetBitmapBits(Bitmap: HBitmap; Count: Longint; Bits: Pointer): Longint;

This function sets the bits of a bitmap to the bit values given by the *lpBits* parameter.

Parameters

hBitmap	**HBITMAP** Identifies the bitmap to be set.
dwCount	**DWORD** Specifies the number of bytes pointed to by *lpBits*.
lpBits	**LPSTR** Points to the bitmap bits that are stored as a long pointer to a byte array.

Return value The return value specifies the number of bytes used in setting the bitmap bits. It is zero if the function fails.

SetBitmapDimension

Syntax DWORD SetBitmapDimension(hBitmap, X, Y)
function SetBitmapDimension(Bitmap: HBitmap; X, Y: Integer): Longint;

This function assigns a width and height to a bitmap in 0.1-millimeter units. These values are not used internally by GDI; the **GetBitmapDimension** function can be used to retrieve them.

Parameters

hBitmap	**HANDLE** Identifies the bitmap.
X	**int** Specifies the width of the bitmap (in 0.1-millimeter units).
Y	**int** Specifies the height of the bitmap (in 0.1-millimeter units).

Return value The return value specifies the previous bitmap dimensions. Height is in the high-order word, and width is in the low-order word.

SetBkColor

Syntax DWORD SetBkColor(hDC, crColor)
function SetBkColor(DC: HDC; Color: TColorRef): Longint;

This function sets the current background color to the color specified by the *crColor* parameter, or to the nearest physical color if the device cannot represent an RGB color value specified by *crColor*.

If the background mode is OPAQUE, GDI uses the background color to fill the gaps between styled lines, gaps between hatched lines in brushes, and character cells. GDI also uses the background color when converting bitmaps from color to monochrome and vice versa.

The background mode is set by the **SetBkMode** function. See the **BitBlt** and **StretchBlt** functions, in this chapter, for color-bitmap conversions.

Parameters

hDC	**HDC** Identifies the device context.
crColor	**COLORREF** Specifies the new background color.

Return value The return value specifies the previous background color as an RGB color value. If an error occurs, the return value is 0x80000000.

SetBkMode

Syntax int SetBkMode(hDC, nBkMode)
function SetBkMode(DC: HDC; BkMode: Integer): Integer;

This function sets the background mode used with text and line styles. The background mode defines whether or not GDI should remove existing background colors on the device surface before drawing text, hatched brushes, or any pen style that is not a solid line.

Parameters

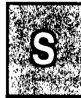
hDC **HDC** Identifies the device context.

nBkMode **int** Specifies the background mode. It can be either one of the following modes:

Value	**Meaning**
OPAQUE	Background is filled with the current background color before the text, hatched brush, or pen is drawn.
TRANSPARENT	Background remains untouched.

Return value The return value specifies the previous background mode. It can be either OPAQUE or TRANSPARENT.

SetBrushOrg



Syntax DWORD SetBrushOrg(hDC, X, Y)
function SetBrushOrg(DC: HDC; X, Y: Integer): Longint;

This function sets the origin of the brush currently selected into the given device context.

Parameters

hDC **HDC** Identifies the device context.

X **int** Specifies the *x*-coordinate (in device units) of the new origin. This value must be in the range 0–7.

Y **int** Specifies the *y*-coordinate (in device units) of the new origin. This value must be in the range 0–7.

Return value The return value specifies the origin of the brush. The previous *x*-coordinate is in the low-order word, and the previous *y*-coordinate is in the high-order word.

Comments The original brush origin is at the coordinate (0,0).

The **SetBrushOrg** function should not be used with stock objects.

SetCapture

Syntax HWND SetCapture(hWnd)
function SetCapture(Wnd: HWnd): HWnd;

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

Parameters *hWnd* **HWND** Identifies the window that is to receive the mouse input.

Return value The return value identifies the window that previously received all mouse input. It is NULL if there is no such window.

Comments When the window no longer requires all mouse input, the application should call the **ReleaseCapture** function so that other windows can receive mouse input.

SetCaretBlinkTime

Syntax void SetCaretBlinkTime(wMSeconds)
procedure SetCaretBlinkTime(MSeconds: Word);

This function sets the caret blink rate (elapsed time between caret flashes) to the number of milliseconds specified by the *wMSeconds* parameter. The caret flashes on or off each *wMSeconds* milliseconds. This means one complete flash (on-off-on) takes 2 × *wMSeconds* milliseconds.

Parameters *wMSeconds* **WORD** Specifies the new blink rate (in milliseconds).

Return value None.

Comments The caret is a shared resource. A window should set the caret blink rate only if it owns the caret. It should restore the previous rate before it loses the input focus or becomes inactive.

SetCaretPos

Syntax void SetCaretPos(X, Y)
procedure SetCaretPos(X, Y: Integer);

This function moves the caret to the position given by logical coordinates specified by the *X* and *Y* parameters. Logical coordinates are relative to the client area of the window that owns them and are affected by the window's mapping mode, so the exact position in pixels depends on this mapping mode.

The **SetCaretPos** function moves the caret only if it is owned by a window in the current task. **SetCaretPos** moves the caret whether or not the caret is hidden.

Parameters

X	**int** Specifies the new *x*-coordinate (in logical coordinates) of the caret.
Y	**int** Specifies the new *y*-coordinate (in logical coordinates) of the caret.

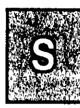
Return value None.

Comments The caret is a shared resource. A window should not move the caret if it does not own the caret.

SetClassLong

Syntax LONG SetClassLong(hWnd, nIndex, dwNewLong)
function SetClassLong(Wnd: HWnd; Index: Integer; NewLong: Longint): Longint;

This function replaces the long value specified by the *nIndex* parameter in the **WNDCLASS** data structure of the window specified by the *hWnd* parameter.



Parameters

hWnd **HWND** Identifies the window.

nIndex **int** Specifies the byte offset of the word to be changed. It can also be one of the following values:

Value	Meaning
GCL_MENUNAME	Sets a new long pointer to the menu name.
GCL_WNDPROC	Sets a new long pointer to the window function.

dwNewLong **DWORD** Specifies the replacement value.

Return value The return value specifies the previous value of the specified long integer.

Comments If the **SetClassLong** function and GCL_WNDPROC index are used to set a window function, the given function must have the window-function

form and be exported in the module-definition file. See the **RegisterClass** function earlier in this chapter for details.

Calling **SetClassLong** with the GCL_WNDPROC index creates a subclass of the window class that affects all windows subsequently created with the class. See Chapter 1, "Window manager interface functions," for more information on window subclassing. An application should not attempt to create a window subclass for standard Windows controls such as combo boxes and buttons.

To access any extra two-byte values allocated when the window-class structure was created, use a positive byte offset as the index specified by the *nIndex* parameter, starting at zero for the first two-byte value in the extra space, 2 for the next two-byte value and so on.

SetClassWord

Syntax WORD SetClassWord(hWnd, nIndex, wNewWord)
function SetClassWord(Wnd: HWnd; Index: Integer; NewWord: Word): Word;

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

Parameters

hWnd **HWND** Identifies the window.

nIndex **int** Specifies the byte offset of the word to be changed. It can also be one of the following values:

Value	Meaning
GCW_CBCLSEXTRA	Sets two new bytes of additional window-class data.
GCW_CBWNDEXTRA	Sets two new bytes of additional window-class data.
GCW_HBRBACKGROUND	Sets a new handle to a background brush.
GCW_HCURSOR	Sets a new handle to a cursor.
GCW_HICON	Sets a new handle to an icon.
GCW_STYLE	Sets a new style bit for the window class.

wNewWord **WORD** Specifies the replacement value.

Return value The return value specifies the previous value of the specified word.

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

To access any extra four-byte values allocated when the window-class structure was created, use a positive byte offset as the index specified by the *nIndex* parameter, starting at zero for the first four-byte value in the extra space, 4 for the next four-byte value and so on.

SetClipboardData

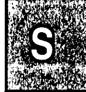
Syntax HANDLE SetClipboardData(wFormat, hMem)
function SetClipboardData(Format: Word; Mem: THandle): THandle;

This function sets a data handle to the clipboard for the data specified by the *hMem* parameter. The data are assumed to have the format specified by the *wFormat* parameter. After setting a clipboard data handle, the **SetClipboardData** function frees the block of memory identified by *hMem*.

Parameters

wFormat **WORD** Specifies a data format. It can be any one of the predefined formats given in Table 4.13, "Predefined data formats."

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



hMem **HANDLE** Identifies the global memory block that contains the data in the specified format. The *hMem* parameter can be NULL. When *hMem* is NULL the application does not have to format the data and provide a handle to it until requested to do so through a WM_RENDERFORMAT message.

Return value The return value identifies the data and is assigned by the clipboard.

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

Table 4.13 shows the various predefined data-format values for the *wFormat* parameter:

Table 4.13
Predefined data formats

Value	Meaning
CF_BITMAP	A handle to a bitmap (HBITMAP).
CF_DIB	A memory block containing a **BITMAPINFO** data structure followed by the bitmap bits.
CF_DIF	Software Arts' Data Interchange Format.
CF_DSPBITMAP	Bitmap display format associated with private format. The *hMem* parameter must be a handle to data that can be displayed in bitmap format in lieu of the privately formatted data.
CF_DSPMETAFILEPICT	Metafile-picture display format associated with private format. The *hMem* parameter must be a handle to data that can be displayed in metafile-picture format in lieu of the privately formatted data.
CF_DSPTEXT	Text display format associated with private format. The *hMem* parameter must be a handle to data that can be displayed in text format in lieu of the privately formatted data.
CF_METAFILEPICT	Handle to a metafile picture format as defined by the **METAFILEPICT** data structure. When passing a CF_METAFILEPICT handle via DDE, the application responsible for deleting *hData* should also free the metafile referred to by the CF_METAFILEPICT handle.
CF_OEMTEXT	Text format contining characters in the OEM character set. Each line ends with a carriage return/linefeed (CR-LF) combination. A null character signals the end of the data.
CF_OWNERDISPLAY	Owner display format. The clipboard owner must display and update the clipboard application window, and will receive WM_ASKCBFORMATNAME, WM_HSCROLLCLIPBOARD, WM_PAINTCLIPBOARD, WM_SIZECLIPBOARD, and WM_VSCROLLCLIPBOARD messages. The *hMem* parameter must be NULL.
CF_PALETTE	Handle to a color palette. Whenever an application places data in the clipboard that depends on or assumes a color palette, it should also place the palette in the clipboard as well. If the clipboard contains data in the CF_PALETTE (logical color palette) format, the application should assume that any other data in the clipboard is realized against that logical palette. The clipboard-viewer application (CLIPBRD.EXE) always uses as its current palette any object in CF_PALETTE format that is in the clipboard when it displays the other formats in the clipboard.
CF_PRIVATEFIRST to CF_PRIVATELAST	Range of integer values used for private formats. Data handles associated with formats in this range will not be freed automatically; any data handles must be freed by the application before the application

Table 4.13: Predefined data formats (continued)

	terminates or when a WM_DESTROYCLIPBOARD message is received.
CF_SYLK	Microsoft Symbolic Link (SYLK) format.
CF_TEXT	Text format. Each line ends with a carriage return/linefeed (CR-LF) combination. A null character signals the end of the data.
CF_TIFF	Tag Image File Format.

Windows supports two formats for text, CF_TEXT and CF_OEMTEXT. CF_TEXT is the default Windows text clipboard format, while Windows uses the CF_OEMTEXT format for text in non-Windows applications. If you call **GetClipboardData** to retrieve data in one text format and the other text format is the only available text format, Windows automatically converts the text to the requested format before supplying it to your application.

An application registers other standard formats, such as Rich Text Format (RTF), by name using the **RegisterClipboardFormat** function rather than by a symbolic constant. For information on these external formats, see the README.TXT file.

SetClipboardViewer

Syntax HWND SetClipboardViewer(hWnd)
function SetClipboardViewer(Wnd: HWnd): HWnd;

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

Parameters *hWnd* **HWND** Identifies the window to receive clipboard-viewer chain messages.

Return value The return value identifies the next window in the clipboard-viewer chain. This handle should be saved in static memory and used in responding to clipboard-viewer chain messages.

Comments Windows that are part of the clipboard-viewer chain must respond to WM_CHANGECBCHAIN, WM_DRAWCLIPBOARD, and WM_DESTROY messages.

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

SetCommBreak

Syntax int SetCommBreak(nCid)
function SetCommBreak(Cid: Integer): Integer;

This function suspends character transmission and places the transmission line in a break state until the **ClearCommBreak** function is called.

Parameters

nCid	**int** Specifies the communication device to be suspended. The **OpenComm** function returns this value.

Return value The return value specifies the result of the function. It is zero if the function is successful. It is negative if *nCid* does not specify a valid device.

SetCommEventMask

Syntax WORD FAR * SetCommEventMask(nCid, nEvtMask)
function SetCommEventMask(Cid: Integer; EvtMask: Word): PWord;

This function enables and retrieves the event mask of the communication device specified by the *nCid* parameter. The bits of the *nEvtMask* parameter define which events are to be enabled. The return value points to the current state of the event mask.

Parameters

nCid	**int** Specifies the communication device to be enabled. The **OpenComm** function returns this value.
nEvtMask	**int** Specifies which events are to be enabled. It can be any combination of the values shown in Table 4.14, "Event values."

Return value The return value points to an integer event mask. Each bit in the event mask specifies whether or not a given event has occurred. A bit is 1 if the event has occurred.

Comments Table 4.14 lists the event values for the *nEvtMask* parameter:

Table 4.14 Event values

Parameter	Type/Description
EV_BREAK	Sets when a break is detected on input.
EV_CTS	Sets when the clear-to-send (CTS) signal changes state.
EV_DSR	Sets when the data-set-ready (DSR) signal changes state.
EV_ERR	Sets when a line-status error occurs. Line-status errors are CE_FRAME, CE_OVERRUN, and CE_RXPARITY.
EV_PERR	Sets when a printer error is detected on a parallel device. Errors are CE_DNS, CE_IOE, CE_LOOP, and CE_PTO.
EV_RING	Sets when a ring indicator is detected.

Table 4.14: Event values (continued)

EV_RLSD	Sets when the receive-line-signal-detect (RLSD) signal changes state.
EV_RXCHAR	Sets when any character is received and placed in the receive queue.
EV_RXFLAG	Sets when the event character is received and placed in the receive queue. The event character is specified in the device's control block.
EV_TXEMPTY	Sets when the last character in the transmit queue is sent.

SetCommState

Syntax int SetCommState(lpDCB)
function SetCommState(var DCB: TDCB): Integer;

This function sets a communication device to the state specified by the device control block pointed to by the *lpDCB* parameter. The device to be set must be identified by the **Id** field of the control block.

This function reinitializes all hardware and controls as defined by *lpDCB*, but does not empty transmit or receive queues.

Parameters *lpDCB* **DCB FAR *** Points to a **DCB** data structure that contains the desired communications setting for the device.

Return value The return value specifies the outcome of the function. It is zero if the function is successful. It is negative if an error occurs.

SetCursor

Syntax HCURSOR SetCursor(hCursor)
function SetCursor(Cursor: HCursor): HCursor;

This function sets the cursor shape to the shape specified by the *hCursor* parameter. The cursor is set only if the new shape is different from the current shape. Otherwise, the function returns immediately. The **SetCursor** function is quite fast if the cursor identified by the *hCursor* parameter is the same as the current cursor.

If *hCursor* is NULL, the cursor is removed from the screen.

Parameters *hCursor* **HCURSOR** Identifies the cursor resource. The resource must have been loaded previously by using the **LoadCursor** function.

Return value The return value identifies the cursor resource that defines the previous cursor shape. It is NULL if there is no previous shape.

Comments The cursor is a shared resource. A window that uses the cursor should set the shape only when the cursor is in its client area or when it is capturing all mouse input. In systems without a mouse, the window should restore the previous cursor shape before the cursor leaves the client area or before the window relinquishes control to another window.

Any application that needs to change the shape of the cursor while it is in a window must make sure the class cursor for the given window's class is set to NULL. If the class cursor is not NULL, Windows restores the previous shape each time the mouse is moved.

The cursor is not shown on the screen if the cursor display count is less than zero. This results from the **HideCursor** function being called more times than the **ShowCursor** function.

SetCursorPos

Syntax void SetCursorPos(X, Y)
procedure SetCursorPos(X, Y: Integer);

This function moves the cursor to the screen coordinates given by the *X* and *Y* parameters. If the new coordinates are not within the screen rectangle set by the most recent **ClipCursor** function, Windows automatically adjusts the coordinates so that the cursor stays within the rectangle.

Parameters

X	**int** Specifies the new *x*-coordinate (in screen coordinates) of the cursor.
Y	**int** Specifies the new *y*-coordinate (in screen coordinates) of the cursor.

Return value None.

Comments The cursor is a shared resource. A window should move the cursor only when the cursor is in its client area.

SetDIBits 3.0

Syntax int SetDIBits(hDC, hBitmap, nStartScan, nNumScans, lpBits, lpBitsInfo, wUsage)

function SetDIBits(DC: HDC; Bitmap: THandle; StartScan, NumScans: Word; Bits: Pointer; var BitsInfo: TBitmapInfo; Usage: Word): Integer;

This function sets the bits of a bitmap to the values given in a device-independent bitmap (DIB) specification.

Parameters

hDC — **HDC** Identifies the device context.

hBitmap — **HBITMAP** Identifies the bitmap.

nStartScan — **WORD** Specifies the scan number of the first scan line in the *lpBits* buffer.

nNumScans — **WORD** Specifies the number of scan lines in the *lpBits* buffer and the number of lines to set in the bitmap identified by the *hBitmap* parameter.

lpBits — **LPSTR** Points to the device-independent bitmap bits that are stored as an array of bytes. The format of the bitmap values depends on the **biBitCount** field of the **BITMAPINFO** structure identified by *lpBitsInfo*. See the description of the **BITMAPINFO** data structure in Chapter 7, "Data types and structures," in *Reference, Volume 2*, for more information.

lpBitsInfo — **LPBITMAPINFO** Points to a **BITMAPINFO** data structure that contains information about the device-independent bitmap.

wUsage — **WORD** Specifies whether the **bmiColors[ ]** fields of the *lpBitsInfo* parameter contain explicit RGB values or indexes into the currently realized logical palette. The *wUsage* parameter must be one of the following values:

Value	Meaning
DIB_PAL_COLORS	The color table consists of an array of 16-bit indexes into the currently realized logical palette.
DIB_RGB_COLORS	The color table contains literal RGB values.

Return value

The return value specifies the number of scan lines successfully copied. It is zero if the function fails.

Comments

The bitmap identified by the *hBitmap* parameter must not be selected into a device context when the application calls this function.

The origin for device-independent bitmaps is the bottom-left corner of the bitmap, not the top-left corner, which is the origin when the mapping mode is MM_TEXT.

This function also accepts a bitmap specification formatted for Microsoft OS/2 Presentation Manager versions 1.1 and 1.2 if the *lpBitsInfo* parameter points to a **BITMAPCOREINFO** data structure.

SetDIBitsToDevice 3.0

Syntax WORD SetDIBitsToDevice(hDC, DestX, DestY, nWidth, nHeight, SrcX, SrcY, nStartScan, nNumScans, lpBits, lpBitsInfo, wUsage)
function SetDIBitsToDevice(DC: HDC; DestX, DestY, Width, Height, SrcX, SrcY, rStartScan, NumScans: Word; Bits: Pointer; var BitsInfo: TBitmapInfo; Usage: Word): Integer;

This function sets bits from a device-independent bitmap (DIB) directly on a device surface. The *SrcX*, *SrcY*, *nWidth*, and *nHeight* parameters define a rectangle within the total DIB. **SetDIBitsToDevice** sets the bits in this rectangle directly on the display surface of the output device identified by the **hDC** parameter, at the location described by the *DestX* and *DestY* parameters.

To reduce the amount of memory required to set bits from a large DIB on a device surface, an application can band the output by repeatedly calling **SetDIBitsToDevice**, placing a different portion of the entire DIB into the *lpBits* buffer each time. The values of the *nStartScan* and *nNumScans* parameters identify the portion of the entire DIB which is contained in the *lpBits* buffer.

Parameters

hDC	**HDC** Identifies the device context.
DestX	**WORD** Specifies the *x*-coordinate of the origin of the destination rectangle.
DestY	**WORD** Specifies the *y*-coordinate of the origin of the destination rectangle.
nWidth	**WORD** Specifies the *x*-extent of the rectangle in the DIB.
nHeight	**WORD** Specifies the *y*-extent of the rectangle in the DIB.
SrcX	**WORD** Specifies the *x*-coordinate of the source in the DIB.
SrcY	**WORD** Specifies the *y*-coordinate of the source in the DIB.
nStartScan	**WORD** Specifies the scan-line number of the DIB which is contained in the first scan line of the *lpBits* buffer.
nNumScans	**WORD** Specifies the number of scan lines of the DIB which are contained in the *lpBits* buffer.

lpBits	**LPSTR** Points to the DIB bits that are stored as an array of bytes.
lpBitsInfo	**LPBITMAPINFO** Points to a **BITMAPINFO** data structure that contains information about the DIB.
wUsage	**WORD** Specifies whether the **bmiColors[ ]** fields of the *lpBitsInfo* parameter contain explicit RGB values or indexes into the currently realized logical palette. The *wUsage* parameter must be one of the following values:

Value	Meaning
DIB_PAL_COLORS	The color table consists of an array of 16-bit indexes into the currently realized logical palette.
DIB_RGB_COLORS	The color table contains literal RGB values.

Return value The return value is the number of scan lines set.

Comments All coordinates are device coordinates (that is, the coordinates of the DIB) except *destX* and *destY*, which are logical coordinates.

The origin for device-independent bitmaps is the bottom-left corner of the DIB, not the top-left corner, which is the origin when the mapping mode is MM_TEXT. This function also accepts a device-independent bitmap specification formatted for Microsoft OS/2 Presentation Manager versions 1.1 and 1.2 if the *lpBitsInfo* parameter points to a **BITMAPCOREINFO** data structure.

SetDlgItemInt

Syntax void SetDlgItemInt(hDlg, nIDDlgItem, wValue, bSigned)
procedure SetDlgItemInt(Dlg: HWnd; IDDlgItem: Integer; Value: Word; Signed: Bool);

This function sets the text of a control in the given dialog box to the string that represents the integer value given by the *wValue* parameter. The **SetDlgItemInt** function converts *wValue* to a string that consists of decimal digits, and then copies the string to the control. If the *bSigned* parameter is nonzero, *wValue* is assumed to be signed. If *wValue* is signed and less than zero, the function places a minus sign before the first digit in the string.

SetDlgItemInt sends a WM_SETTEXT message to the given control.

Parameters *hDlg* **HWND** Identifies the dialog box that contains the control.

	nIDDlgItem	**int** Specifies the control to be modified.
	wValue	**WORD** Specifies the value to be set.
	bSigned	**BOOL** Specifies whether or not the integer value is signed.
Return value	None.	

SetDlgItemText

Syntax void SetDlgItemText(hDlg, nIDDlgItem, lpString)
procedure SetDlgItemText(Dlg: HWnd; IDDlgItem: Integer; Str: PChar);

This function sets the caption or text of a control in the dialog box specified by the *hDlg* parameter. The **SetDlgItemText** function sends a WM_SETTEXT message to the given control.

Parameters	*hDlg*	**HWND** Identifies the dialog box that contains the control.
	nIDDlgItem	**int** Specifies the control whose text is to be set.
	lpString	**LPSTR** Points to the null-terminated character string that is to be copied to the control.

Return value None.

SetDoubleClickTime

Syntax void SetDoubleClickTime(wCount)
procedure SetDoubleClickTime(Count: Word);

This function sets the double-click time for the mouse. A double-click is a series of two clicks of the mouse button, the second occurring within a specified time after the first. The double-click time is the maximum number of milliseconds that may occur between the first and second clicks of a double-click.

Parameters	*wCount*	**WORD** Specifies the number of milliseconds that can occur between double-clicks.

Return value None.

Comments If the *wCount* parameter is set to zero, Windows will use the default double-click time of 500 milliseconds.

The **SetDoubleClickTime** function alters the double-click time for all windows in the system.

SetEnvironment

Syntax

int SetEnvironment(lpPortName, lpEnviron, nCount)
function SetEnvironment(PortName: PChar; Environ: Pointer; Count: Word): Integer;

This function copies the contents of the buffer specified by the *lpEnviron* parameter into the environment associated with the device attached to the system port specified by the *lpPortName* parameter. The **SetEnvironment** function deletes any existing environment. If there is no environment for the given port, **SetEnvironment** creates one. If the *nCount* parameter is zero, the existing environment is deleted and not replaced.

Parameters

lpPortName	**LPSTR** Points to a null-terminated character string that specifies the name of the desired port.
lpEnviron	**LPSTR** Points to the buffer that contains the new environment.
nCount	**WORD** Specifies the number of bytes to be copied.

Return value

The return value specifies the actual number of bytes copied to the environment. It is zero if there is an error. It is –1 if the environment is deleted.

Comments

The first field in the buffer pointed to by the *lpEnviron* parameter must be the same as that passed in the *lpDeviceName* parameter of the **CreateDC** function. If *lpPortName* specifies a null port (as defined in the WIN.INI file), the device name pointed to by *lpEnviron* is used to locate the desired environment.

SetErrorMode

Syntax

WORD SetErrorMode (wMode)
function SetErrorMode(Mode: Word): Word;

This function controls whether Windows handles DOS Function 24H errors or allows the calling application to handle them.

Windows intercepts all INT 24H errors. If the application calls **SetErrorMode** with the *wMode* parameter set to zero and an INT 24H error subsequently occurs, Windows displays an error message box. If the application calls **SetErrorMode** with *wMode* set to 1 and an INT 24H occurs, Windows does not display the standard INT 24H error message box, but rather fails the original INT 21H call back to the application. This

allows the application to handle disk errors using INT 21H, AH=59H (**Get Extended Error**) as appropriate.

Parameters *wMode* **WORD** Specifies the error mode flag. If bit 0 is set to zero, Windows displays an error message box when an INT 24H error occurs. If bit 0 is set to 1, Windows fails the INT 21H call to the calling application and does not display a message box.

Return value The return value specifies the previous value of the error mode flag.

SetFocus

Syntax HWND SetFocus(hWnd)
function SetFocus(Wnd: HWnd): HWnd;

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

The **SetFocus** function sends a WM_KILLFOCUS message to the window that loses the input focus and a WM_SETFOCUS message to the window that receives the input focus. It also activates either the window that receives the focus or the parent of the window that receives the focus.

Parameters *hWnd* **HWND** Identifies the window to receive the keyboard input.

Return value The return value identifies the window that previously had the input focus. It is NULL if there is no such window.

Comments If a window is active but doesn't have the focus (that is, no window has the focus), any key pressed will produce the WM_SYSCHAR, WM_SYSKEYDOWN, or WM_SYSKEYUP message. If the VK_MENU key is also pressed, the *lParam* parameter of the message will have bit 30 set. Otherwise, the messages that are produced do *not* have this bit set.

SetHandleCount 3.0

Syntax WORD SetHandleCount(wNumber)
function SetHandleCount(Number: Word): Word;

This function changes the number of file handles available to a task. By default, the maximum number of file handles available to a task is 20.

Parameters *wNumber* **WORD** Specifies the number of file handles needed by the application. The maximum is 255.

Return value The return value specifies the number of file handles actually available to the application. It may be less than the number specified by the *wNumber* parameter.

SetKeyboardState

Syntax void SetKeyboardState(lpKeyState)
procedure SetKeyboardState(var KeyState: TKeyboardState);

This function copies the 256 bytes pointed to by the *lpKeyState* parameter into the Windows keyboard-state table.

Parameters *lpKeyState* **BYTE FAR** * Points to an array of 256 bytes that contains keyboard key states.

Return value None.

Comments In many cases, an application should call the **GetKeyboardState** function first to initialize the 256-byte array. The application should then change the desired bytes.

SetKeyboardState sets the LEDs and BIOS flags for the NUMLOCK, CAPSLOCK, and SCROLL LOCK keys according to the toggle state of the VK_NUMLOCK, VK_CAPITAL, and VK_OEM_SCROLL entries of the array.

For more information, see the description of **GetKeyboardState**, earlier in this chapter.

S

SetMapMode

Syntax int SetMapMode(hDC, nMapMode)
function SetMapMode(DC: HDC; MapMode: Integer): Integer;

This function sets the mapping mode of the specified device context. The mapping mode defines the unit of measure used to transform logical units into device units, and also defines the orientation of the device's *x*- and *y*-axes. GDI uses the mapping mode to convert logical coordinates into the appropriate device coordinates.

Parameters *hDC* **HDC** Identifies the device context.

nMapMode **int** Specifies the new mapping mode. It can be any one of the values shown in Table 4.15, "Mapping modes."

Return value The return value specifies the previous mapping mode.

Comments The MM_TEXT mode allows applications to work in device pixels, whose size varies from device to device.

The MM_HIENGLISH, MM_HIMETRIC, MM_LOENGLISH, MM_LOMETRIC, and MM_TWIPS modes are useful for applications that need to draw in physically meaningful units (such as inches or millimeters).

The MM_ISOTROPIC mode ensures a 1:1 aspect ratio, which is useful when preserving the exact shape of an image is important.

The MM_ANISOTROPIC mode allows the *x*- and *y*-coordinates to be adjusted independently.

Table 4.15 shows the value and meaning of the various mapping modes:

Table 4.15 Mapping modes

Value	Meaning
MM_ANISOTROPIC	Logical units are mapped to arbitrary units with arbitrarily scaled axes. The **SetWindowExt** and **SetViewportExt** functions must be used to specify the desired units, orientation, and scaling.
MM_HIENGLISH	Each logical unit is mapped to 0.001 inch. Positive *x* is to the right; positive *y* is up.
MM_HIMETRIC	Each logical unit is mapped to 0.01 millimeter. Positive *x* is to the right; positive *y* is up.
MM_ISOTROPIC	Logical units are mapped to arbitrary units with equally scaled axes; that is, one unit along the *x*-axis is equal to one unit along the *y*-axis. The **SetWindowExt** and **SetViewportExt** functions must be used to specify the desired units and the orientation of the axes. GDI makes adjustments as necessary to ensure that the *x* and *y* units remain the same size.
MM_LOENGLISH	Each logical unit is mapped to 0.01 inch. Positive *x* is to the right; positive *y* is up.
MM_LOMETRIC	Each logical unit is mapped to 0.1 millimeter. Positive *x* is to the right; positive *y* is up.
MM_TEXT	Each logical unit is mapped to one device pixel. Positive *x* is to the right; positive *y* is down.
MM_TWIPS	Each logical unit is mapped to one twentieth of a printer's point (1/1440 inch). Positive *x* is to the right; positive *y* is up.

SetMapperFlags

Syntax DWORD SetMapperFlags(hDC, dwFlag)
function SetMapperFlags(DC: HDC; Flag: Longint): Longint;

This function alters the algorithm that the font mapper uses when it maps logical fonts to physical fonts. When the first bit of the *wFlag* parameter is set to 1, the mapper will only select fonts whose *x*-aspect and *y*-aspect exactly match those of the specified device. If no fonts exist with a matching aspect height and width, GDI chooses an aspect height and width and selects fonts with aspect heights and widths that match the one chosen by GDI.

Parameters

hDC **HDC** Identifies the device context that contains the font-mapper flag.

dwFlag **DWORD** Specifies whether the font mapper attempts to match a font's aspect height and width to the device. When the first bit is set to 1, the mapper will only select fonts whose *x*-aspect and *y*-aspect exactly match those of the specified device.

Return value The return value specifies the previous value of the font-mapper flag.

Comments The remaining bits of the *dwFlag* parameter must be zero.

SetMenu

Syntax BOOL SetMenu(hWnd, hMenu)
function SetMenu(Wnd: HWnd; Menu: HMenu): Bool;

This function sets the given window's menu to the menu specified by the *hMenu* parameter. If *hMenu* is NULL, the window's current menu is removed. The **SetMenu** function causes the window to be redrawn to reflect the menu change.

Parameters

hWnd **HWND** Identifies the window whose menu is to be changed.

hMenu **HMENU** Identifies the new menu.

Return value The return value specifies whether the menu is changed. It is nonzero if the menu is changed. Otherwise, it is zero.

Comments **SetMenu** will not destroy a previous menu. An application should call the **DestroyMenu** function to accomplish this task.

SetMenuItemBitmaps 3.0

Syntax

BOOL SetMenuItemBitmaps(hMenu, nPosition, wFlags, hBitmapUnchecked, hBitmapChecked)
function SetMenuItemBitmaps(Menu: HMenu; Position, Flags: Word; BitmapUnchecked, BitmapChecked: HBitmap): Bool;

This function associates the specified bitmaps with a menu item. Whether the menu item is checked or unchecked, Windows displays the appropriate bitmap next to the menu item.

Parameters

hMenu	**HMENU** Identifies the menu to be changed.
nPosition	**WORD** Specifies the menu item to be changed. If *wFlags* is set to MF_BYPOSITION, *nPosition* specifies the position of the menu item; the first item in the menu is at position 0. If *wFlags* is set to MF_BYCOMMAND, then *nPosition* specifies the command ID of the menu item.
wFlags	**WORD** Specifies how the *nPosition* parameter is interpreted. It may be set to MF_BYCOMMAND (the default) or MF_BYPOSITION.
hBitmapUnchecked	**HBITMAP** Identifies the bitmap to be displayed when the menu item is not checked.
hBitmapChecked	**HBITMAP** Identifies the bitmap to be displayed when the menu item is checked.

Return value

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

Comments

If either the *hBitmapUnchecked* or the *hBitmapChecked* parameters is NULL, then Windows displays nothing next to the menu item for the corresponding attribute. If both parameters are NULL, Windows uses the default checkmark when the item is checked and removes the checkmark when the item is unchecked.

When the menu is destroyed, these bitmaps are not destroyed; it is the responsibility of the application to destroy them.

The **GetMenuCheckMarkDimensions** function retrieves the dimensions of the default checkmark used for menu items. The application should use these values to determine the appropriate size for the bitmaps supplied with this function.

SetMessageQueue

Syntax BOOL SetMessageQueue(cMsg)
function SetMessageQueue(Msg: Integer): Bool;

This function creates a new message queue. It is particularly useful in applications that require a queue that contains more than eight messages (the maximum size of the default queue). The *cMsg* parameter specifies the size of the new queue; the function must be called from an application's WinMain function before any windows are created and before any messages are sent. The **SetMessageQueue** function destroys the old queue, along with messages it might contain.

Parameters *cMsg* **int** Specifies the maximum number of messages that the new queue may contain.

Return value The return value specifies whether a new message queue is created. It is nonzero if the function creates a new queue. Otherwise, it is zero.

Comments If the return value is zero, the application has no queue because the **SetMessageQueue** function deletes the original queue before attempting to create a new one. The application must continue calling **SetMessageQueue** with a smaller queue size until the function returns a nonzero value.

SetMetaFileBits

Syntax HANDLE SetMetaFileBits(hMem)
function SetMetaFileBits(Mem: THandle): THandle;

This function creates a memory metafile from the data in the global memory block specified by the *hMem* parameter.

Parameters *hMem* **HANDLE** Identifies the global memory block that contains the metafile data. It is assumed that the data were previously created by using the **GetMetaFileBits** function.

Return value The return value identifies a memory metafile if the function is successful. Otherwise, the return value is NULL.

Comments After the **SetMetaFileBits** function returns, the metafile handle returned by the function should be used instead of the handle identified by the *hMem* parameter to refer to the metafile.

SetPaletteEntries 3.0

Syntax WORD SetPaletteEntries(hPalette, wStartIndex, wNumEntries, lpPaletteEntries)
function SetPaletteEntries(Palette: HPalette; StartIndex, NumEntries: Word; var PaletteEntries): Word;

This function sets RGB color values and flags in a range of entries in a logical palette.

Parameters

hPalette	**HPALETTE** Identifies the logical palette.
wStartIndex	**WORD** Specifies the first entry in the logical palette to be set.
wNumEntries	**WORD** Specifies the number of entries in the logical palette to be set.
lpPaletteEntries	**LPPALETTEENTRY** Points to the first member of an array of **PALETTEENTRY** data structures containing the RGB values and flags.

Return value The return value is the number of entries set in the logical palette. It is zero if the function failed.

Comments If the logical palette is selected into a device context when the application calls **SetPalette-**

Entries, the changes will not take effect until the application calls **RealizePalette**.

SetParent

Syntax HWND SetParent(hWndChild, hWndNewParent)
function SetParent(WndChild, WndNewParent: HWnd): HWnd;

This function changes the parent window of a child window. If the window identified by the *hWndChild* parameter is visible, Windows performs the appropriate redrawing and repainting.

Parameters

hWndChild	**HWND** Identifies the child window.
hWndNewParent	**HWND** Identifies the new parent window.

Return value The return value identifies the previous parent window.

SetPixel

Syntax DWORD SetPixel(hDC, X, Y, crColor)
function SetPixel(DC: HDC; X, Y: Integer; Color: TColorRef): Longint;

This function sets the pixel at the point specified by the *X* and *Y* parameters to the closest approximation of the color specified by the *crColor* parameter. The point must be in the clipping region. If the point is not in the clipping region, the function is ignored.

Parameters

hDC	**HDC** Identifies the device context.
X	**int** Specifies the logical *x*-coordinate of the point to be set.
Y	**int** Specifies the logical *y*-coordinate of the point to be set.
crColor	**COLORREF** Specifies the color used to paint the point.

Return value The return value specifies an RGB color value for the color that the point is actually painted. This value can be different than that specified by the *crColor* parameter if an approximation of that color is used. If the function fails (if the point is outside the clipping region) the return value is –1.

Comments Not all devices support the **SetPixel** function. For more information, see the RC_BITBLT capability in the **GetDeviceCaps** function, earlier in this chapter.

S

SetPolyFillMode

Syntax int SetPolyFillMode(hDC, nPolyFillMode)
function SetPolyFillMode(DC: HDC; PolyFillMode: Integer): Integer;

This function sets the polygon-filling mode for the GDI functions that use the polygon algorithm to compute interior points.

Parameters

hDC **HDC** Identifies the device context.

nPolyFillMode
int Specifies the new filling mode. The *nPolyFillMode* parameter may be either of the following values:

Value	Meaning
ALTERNATE	Selects alternate mode.
WINDING	Selects winding number mode.

Return value The return value specifies the previous filling mode. It is zero if there is an error.

Comments In general, the modes differ only in cases where a complex, overlapping polygon must be filled (for example, a five-sided polygon that forms a five-pointed star with a pentagon in the center). In such cases, ALTERNATE mode fills every other enclosed region within the polygon (that is, the points of the star), but WINDING mode fills all regions (that is, the points and the pentagon).

When the filling mode is ALTERNATE, GDI fills the area between odd-numbered and even-numbered polygon sides on each scan line. That is, GDI fills the area between the first and second side, between the third and fourth side, and so on.

To fill all regions, WINDING mode causes GDI to compute and draw a border that encloses the polygon but does not overlap. For example, in WINDING mode, the five-sided polygon that forms the star is drawn as a ten-sided polygon with no overlapping sides; the resulting star is filled.

SetProp

Syntax BOOL SetProp(hWnd, lpString, hData)
function SetProp(Wnd: HWnd; Str: PChar; Data: THandle): Bool;

This function adds a new entry or changes an existing entry in the property list of the specified window. The **SetProp** function adds a new entry to the list if the character string specified by the *lpString* parameter does not already exist in the list. The new entry contains the string and the handle. Otherwise, the function replaces the string's current handle with the one specified by the *hData* parameter.

The *hData* parameter can contain any 16-bit value useful to the application.

Parameters

hWnd	**HWND** Identifies the window whose property list is to receive the new entry.
lpString	**LPSTR** Points to a null-terminated character string or an atom that identifies a string. If an atom is given, it must have been previously created by using the **AddAtom** function. The atom, a 16-bit value, must be placed in the low-order word of *lpString*; the high-order word must be zero.

hData **HANDLE** Identifies a data handle to be copied to the property list.

Return value The return value specifies the outcome of the function. It is nonzero if the data handle and string are added to the property list. Otherwise, it is zero.

Comments The application is responsible for removing all entries it has added to the property list before destroying the window (that is, before the application processes the WM_DESTROY message). The **RemoveProp** function must be used to remove entries from a property list.

SetRect

Syntax void SetRect(lpRect, X1, Y1, X2, Y2)
procedure SetRect(var Rect: TRect; X1, Y1, X2, Y2: Integer);

This function creates a new rectangle by filling the **RECT** data structure pointed to by the *lpRect* parameter with the coordinates given by the *X1*, *Y1*, *X2*, and *Y2* parameters.

Parameters

lpRect	**LPRECT** Points to the **RECT** data structure that is to receive the new rectangle coordinates.
X1	**int** Specifies the *x*-coordinate of the upper-left corner.
Y1	**int** Specifies the *y*-coordinate of the upper-left corner.
X2	**int** Specifies the *x*-coordinate of the lower-right corner.
Y2	**int** Specifies the *y*-coordinate of the lower-right corner.

Return value None.

Comments The width of the rectangle, specified by the absolute value of *X2* – *X1*, must not exceed 32,767 units. This limit applies to the height of the rectangle as well.

SetRectEmpty

Syntax void SetRectEmpty(lpRect)
procedure SetRectEmpty(var Rect: TRect);

This function creates an empty rectangle (all coordinates equal to zero).

Parameters

lpRect	**LPRECT** Points to the **RECT** data structure that is to receive the empty rectangle.

Return value None.

SetRectRgn

Syntax void SetRectRgn(hRgn, X1, Y1, X2, Y2)
procedure SetRectRgn(Rgn: HRgn; X1, Y1, X2, Y2: Integer);

This function creates a rectangular region. Unlike **CreateRectRegion,** however, it does not call the local memory manager; instead, it uses the space allocated for the region associated with the *hRgn* parameter. The points given by the *X1*, *Y1*, *X2*, and *Y2* parameters specify the minimum size of the allocated space.

Parameters

hRgn	**HANDLE** Identifies the region.
X1	**int** Specifies the *x*-coordinate of the upper-left corner of the rectangular region.
Y1	**int** Specifies the *y*-coordinate of the upper-left corner of the rectangular region.
X2	**int** Specifies the *x*-coordinate of the lower-right corner of the rectangular region.
Y2	**int** Specifies the *y*-coordinate of the lower-right corner of the rectangular region.

Return value None.

Comments Use this function instead of the **CreateRectRgn** function to avoid calls to the local memory manager.

SetResourceHandler

Syntax FARPROC SetResourceHandler(hInstance, lpType, lpLoadFunc)
function SetResourceHandler(Instance: THandle; ResType: Pointer; LoadFunc: TFarProc): TFarProc;

This function sets up a function to load resources. It is used internally by Windows to implement calculated resources. Applications may find this function useful for handling their own resource types, but its use is not required. The *lpLoadFunc* parameter points to an application-supplied callback function. The function pointed to by the *lpLoadFunc* parameter receives information about the resource to be locked and can process that

information as desired. After the function pointed to by *lpLoadFunc* returns, **LockResource** attempts to lock the resource once more.

Parameters

hInstance	**HANDLE** Identifies the instance of the module whose executable file contains the resource.
lpType	**LPSTR** Points to a short integer that specifies a resource type.
lpLoadFunc	**FARPROC** Is the procedure-instance address of the application-supplied callback function. See the following "Comments" section for details.

Return value The return value points to the application-supplied function.

Comments The callback function must use the Pascal calling convention and must be declared **FAR**.

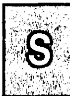
Callback function

FARPROC FAR PASCAL *LoadFunc(hMem, hInstance, hResInfo)*
HANDLE *hMem;*
HANDLE *hInstance;*
HANDLE *hResInfo;*

LoadFunc is a placeholder for the application-supplied function name. The actual name must be exported by including it in an **EXPORTS** statement in the application's module-definition file.

Parameters

hMem	Identifies a stored resource.
hInstance	Identifies the instance of the module whose executable file contains the resource.
hResInfo	Identifies the resource. It is assumed that the resource was created previously by using the **FindResource** function.



Comments The *hMem* parameter is NULL if the resource has not yet been loaded. If an attempt to lock a block specified by *hMem* fails, this means the resource has been discarded and must be reloaded.

The dialog-function address, passed as the *lpLoadFunc* parameter, must be created by using the **MakeProcInstance** function.

SetROP2

Syntax

int SetROP2(hDC, nDrawMode)
function SetROP2(DC: HDC; DrawMode: Integer): Integer;

This function sets the current drawing mode. GDI uses the drawing mode to combine pens and interiors of filled objects with the colors already on the display surface. The mode specifies how the color of the pen or interior and the color already on the display surface yield a new color.

Parameters

hDC **HDC** Identifies the device context.

nDrawMode **int** Specifies the new drawing mode. It can be any one of the values given in Table 4.16, "Drawing modes."

Return value

The return value specifies the previous drawing mode. It can be any one of the values given in Chapter 11, "Binary and ternary raster-operation codes," in *Reference, Volume 2*.

Comments

Drawing modes define how GDI combines source and destination colors when drawing with the current pen. The drawing modes are actually binary raster-operation codes, representing all possible Boolean functions of two variables, using the binary operations AND, OR, and XOR (exclusive OR), and the unary operation NOT. The drawing mode is for raster devices only; it is not available on vector devices. For more information, see the RC_BITBLT capability in the **GetDeviceCaps** function, earlier in this chapter. Table 4.16 shows the value of various drawing modes for the *nDrawMode* parameter:

Table 4.16 Drawing modes

Value	Meaning
R2_BLACK	Pixel is always black.
R2_WHITE	Pixel is always white.
R2_NOP	Pixel remains unchanged.
R2_NOT	Pixel is the inverse of the display color.
R2_COPYPEN	Pixel is the pen color.
R2_NOTCOPYPEN	Pixel is the inverse of the pen color.
R2_MERGEPENNOT	Pixel is a combination of the pen color and the inverse of the display color.
R2_MASKPENNOT	Pixel is a combination of the colors common to both the pen and the inverse of the display.
R2_MERGENOTPEN	Pixel is a combination of the display color and the inverse of the pen color.
R2_MASKNOTPEN	Pixel is a combination of the colors common to both the display and the inverse of the pen.
R2_MERGEPEN	Pixel is a combination of the pen color and the display color.
R2_NOTMERGEPEN	Pixel is the inverse of the R2_MERGEPEN color.

Table 4.16: Drawing modes (continued)

R2_MASKPEN	Pixel is a combination of the colors common to both the pen and the display.
R2_NOTMASKPEN	Pixel is the inverse of the R2_MASKPEN color.
R2_XORPEN	Pixel is a combination of the colors in the pen and in the display, but not in both.
R2_NOTXORPEN	Pixel is the inverse of the R2_XORPEN color.

For more information about the drawing modes, see Chapter 11, "Binary and ternary raster-operation codes," in *Reference, Volume 2*.

SetScrollPos

Syntax

int SetScrollPos(hWnd, nBar, nPos, bRedraw)
function SetScrollPos(Wnd: HWnd; Bar, Pos: Integer; Redraw: Bool): Integer;

This function sets the current position of a scroll-bar thumb to that specified by the *nPos* parameter and, if specified, redraws the scroll bar to reflect the new position.

Parameters

hWnd — **HWND** Identifies the window whose scroll bar is to be set.

nBar — **int** Specifies the scroll bar to be set. It can be one of the following values:

Value	**Meaning**
SB_CTL	Sets the position of a scroll-bar control. In this case, the *hWnd* parameter must be the handle of a scroll-bar control.
SB_HORZ	Sets a window's horizontal scroll-bar position.
SB_VERT	Sets a window's vertical scroll-bar position.

nPos — **int** Specifies the new position. It must be within the scrolling range.

bRedraw — **BOOL** Specifies whether the scroll bar should be redrawn to reflect the new position. If the *bRedraw* parameter is nonzero, the scroll bar is redrawn. If it is zero, it is not redrawn.

Return value

The return value specifies the previous position of the scroll-bar thumb.

Comments

Setting the *bRedraw* parameter to zero is useful whenever the scroll bar will be redrawn by a subsequent call to another function.

SetScrollRange

Syntax void SetScrollRange(hWnd, nBar, nMinPos, nMaxPos, bRedraw)
procedure SetScrollRange(Wnd: HWnd; Bar, MinPos, MaxPos: Integer; Redraw: Bool);

This function sets minimum and maximum position values for the given scroll bar. It can also be used to hide or show standard scroll bars by setting the *nMinPos* and *nMaxPos* parameters to zero.

Parameters

hWnd **HWND** Identifies a window or a scroll-bar control, depending on the value of the *nBar* parameter.

nBar **int** Specifies the scroll bar to be set. It can be one of the following values:

Value	Meaning
SB_CTL	Sets the range of a scroll-bar control. In this case, the *hWnd* parameter must be the handle of a scroll-bar control.
SB_HORZ	Sets a window's horizontal scroll-bar range.
SB_VERT	Sets a window's vertical scroll-bar range.

nMinPos **int** Specifies the minimum scrolling position.

nMaxPos **int** Specifies the maximum scrolling position.

bRedraw **BOOL** Specifies whether or not the scroll bar should be redrawn to reflect the change. If the *bRedraw* parameter is nonzero, the scroll bar is redrawn. If it is zero, it is not redrawn.

Return value None.

Comments An application should not call this function to hide a scroll bar while processing a scroll-bar notification message.

If **SetScrollRange** immediately follows the **SetScrollPos** function, the *bRedraw* parameter in **SetScrollPos** should be set to zero to prevent the scroll bar from being drawn twice. The difference between the values specified by the *nMinPos* and *nMaxPos* parameters must not be greater than 32,767.

SetSoundNoise

Syntax int SetSoundNoise(nSource, nDuration)
function SetSoundNoise(Source, Duration: Integer): Integer;

This function sets the source and duration of a noise in the noise hardware of the play device.

Parameters *nSource* **int** Specifies the noise source. It can be any one of the following values, where N is a value used to derive a target frequency:

Value	Meaning
S_PERIOD512	Source frequency is N/512 (high pitch); hiss is less coarse.
S_PERIOD1024	Source frequency is N/1024.
S_PERIOD2048	Source frequency is N/2048 (low pitch); hiss is coarser.
S_PERIODVOICE	Source frequency from voice channel 3.
S_WHITE512	Source frequency is N/512 (high pitch); hiss is less coarse.
S_WHITE1024	Source frequency is N/1024.
S_WHITE2048	Source frequency is N/2048 (low pitch); hiss is coarser.
S_WHITEVOICE	Source frequency from voice channel 3.

nDuration **int** Specifies the duration of the noise (in clock ticks).

Return value The return value specifies the result of the function. It is zero if the function is successful. If the source is invalid, the return value is S_SERDSR.

SetStretchBltMode

Syntax int SetStretchBltMode(hDC, nStretchMode)
function SetStretchBltMode(DC: HDC; StretchMode: Integer): Integer;

This function sets the stretching mode for the **StretchBlt** function. The stretching mode defines which scan lines and/or columns **StretchBlt** eliminates when contracting a bitmap.

Parameters *hDC* **HDC** Identifies the device context.

nStretchMode **int** Specifies the new stretching mode. It can be one of the following values:

Value	Meaning
BLACKONWHITE	AND in the *eliminated* lines. This mode preserves black pixels at the expense of white pixels by using the AND

	operator on the eliminated lines and those remaining.
COLORONCOLOR	Deletes the *eliminated* lines. This mode deletes all eliminated lines without trying to preserve their information.
WHITEONBLACK	OR in the *eliminated* lines. This mode preserves white pixels at the expense of black pixels by using the OR operator on the lines to be eliminated and the remaining lines.

The BLACKONWHITE and WHITEONBLACK modes are typically used to preserve foreground pixels in monochrome bitmaps. The COLORONCOLOR mode is typically used to preserve color in color bitmaps.

Return value The return value specifies the previous stretching mode. It can be BLACKONWHITE, COLORONCOLOR, or WHITEONBLACK.

SetSwapAreaSize

Syntax LONG SetSwapAreaSize(rsSize)
function SetSwapAreaSize(Size: Word): Longint;

This function increases the amount of memory that an application uses for its code segments. The maximum amount of memory available is one-half of the space remaining after Windows is loaded.

Parameters

rsSize	**WORD** Specifies the number of 16-byte paragraphs requested by the application for use as a code segment.

Return value The low-order word of the return value specifies the number of paragraphs obtained for use as a code segment space (or the current size if *rsSize* is zero); the high-order word specifies the maximum size available.

Comments If *rsSize* specifies a size larger than is available, this function sets the size to the available amount.

Once memory has been dedicated for use as code segment space, an application cannot use it as a data segment by calling the **GlobalAlloc** function.

Calling this function improves an application's performance by helping prevent thrashing. However, it reduces the amount of memory available for data objects and can reduce the performance of other applications.

Before calling **SetSwapAreaSize**, an application should call **GetNumTasks** to determine how many other tasks are running.

SetSysColors

Syntax void SetSysColors(nChanges, lpSysColor, lpColorValues)
procedure SetSysColors(Changes: Integer; var SysColor; var ColorValues);

This function sets the system colors for one or more display elements. Display elements are the various parts of a window and the Windows display that appear on the system display screen. The **SetSysColors** function changes the number of elements specified by the *nChanges* parameter, using the color and system-color index contained in the arrays pointed to by the *lpSysColor* and *lpColorValues* parameters.

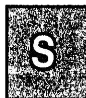
SetSysColors sends a WM_SYSCOLORCHANGE message to all windows to inform them of the change in color. It also directs Windows to repaint the affected portions of all currently visible windows.

Parameters

nChanges **int** Specifies the number of system colors to be changed.

lpSysColor **LPINT** Points to an array of integer indexes that specify the elements to be changed. The index values that can be used are listed in Table 4.17, "System color indexes."

lpColorValues **DWORD FAR *** Points to an array of unsigned long integers that contains the new RGB color values for each element.



Return value None.

Comments **SetSysColors** changes the internal system list only. It does not change the [COLORS] section of the Windows initialization file, WIN.INI. Changes apply to the current Windows session only. System colors are a shared resource. An application should not change a color if it does not wish to change colors for all windows in all currently running applications. System colors for monochrome displays are usually interpreted as various shades of gray. Table 4.17 lists the values for the *lpSysColor* parameter:

Table 4.17 System color indexes

Value	Meaning
COLOR_ACTIVEBORDER	Active window border.
COLOR_ACTIVECAPTION	Active window caption.
COLOR_APPWORKSPACE	Background color of multiple document interface (MDI) applications.
COLOR_BACKGROUND	Desktop.
COLOR_BTNFACE	Face shading on push buttons.
COLOR_BTNSHADOW	Edge shading on push buttons.

Table 4.17: System color indexes (continued)

COLOR_BTNTEXT	Text on push buttons.
COLOR_CAPTIONTEXT	Text in caption, size box, scroll-bar arrow box.
COLOR_GRAYTEXT	Grayed (disabled) text. This color is set to 0 if the current display driver does not support a solid gray color.
COLOR_HIGHLIGHT	Items selected item in a control.
COLOR_HIGHLIGHTTEXT	Text of item selected in a control.
COLOR_INACTIVEBORDER	Inactive window border.
COLOR_INACTIVECAPTION	Inactive window caption.
COLOR_MENU	Menu background.
COLOR_MENUTEXT	Text in menus.
COLOR_SCROLLBAR	Scroll-bar gray area.
COLOR_WINDOW	Window background.
COLOR_WINDOWFRAME	Window frame.
COLOR_WINDOWTEXT	Text in windows.

SetSysModalWindow

Syntax HWND SetSysModalWindow(hWnd)
function SetSysModalWindow(Wnd: HWnd): HWnd;

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

Parameters *hWnd* **HWND** Identifies the window to be made system modal.

Return value The return value identifies the window that was previously the system-modal window.

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

SetSystemPaletteUse 3.0

Syntax WORD SetSystemPaletteUse(hDC, wUsage)
function SetSystemPaletteUse(DC: HDC; Usage: Word): Word;

This function allows an application to use the full system palette. By default, the system palette contains 20 static colors which are not changed when an application realizes its logical palette. The device context identified by the *hDC* parameter must refer to a device that supports color palettes.

Parameters

hDC — **HDC** Identifies the device context.

wUsage — **WORD** Specifies the new use of the system palette. It can be either of these values:

Value	Meaning
SYSPAL_NOSTATIC	System palette contains no static colors except black and white.
SYSPAL_STATIC	System palette contains static colors which will not change when an application realizes its logical palette.

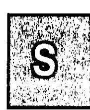
Return value The return value specifies the previous usage of the system palette. It is either SYSPAL_NOSTATIC or SYSPAL_STATIC.

Comments An application must call this function only when its window has input focus.

If an application calls **SetSystemPaletteUse** with *wUsage* set to SYSPAL_NOSTATIC, Windows continues to set aside two entries in the system palette for pure white and pure black, respectively.

After calling this function with *wUsage* set to SYSPAL_NOSTATIC, an application must follow these steps:

1. Call **UnrealizeObject** to force GDI to remap the logical palette completely when it is realized.
2. Realize the logical palette.
3. Call **GetSysColors** to save the current system-color settings.
4. Call **SetSysColors** to set the system colors to reasonable values using black and white. For example, adjacent or overlapping items (such as window frames and borders) should be set to black and white, respectively.
5. Broadcast the WM_SYSCOLORCHANGE message to allow other windows to be redrawn with the new system colors.



When the application's window loses focus or closes, the application must perform the following steps:

1. Call **SetSystemPaletteUse** with the *wUsage* parameter set to SYSPAL_STATIC.
2. Call **UnrealizeObject** to force GDI to remap the logical palette completely when it is realized.

3. Realize the logical palette.
4. Restore the system colors to their previous values.
5. Broadcast the WM_SYSCOLORCHANGE message.

SetTextAlign

Syntax WORD SetTextAlign(hDC, wFlags)
function SetTextAlign(DC: HDC; Flags: Word): Word;

This function sets the text-alignment flags for the given device context. The **TextOut** and **ExtTextOut** functions use these flags when positioning a string of text on a display or device. The flags specify the relationship between a specific point and a rectangle that bounds the text. The coordinates of this point are passed as parameters to the **TextOut** function. The rectangle that bounds the text is formed by the adjacent character cells in the text string.

Parameters

hDC — **HDC** Identifies the device or display selected for text output.

wFlags — **WORD** Specifies a mask of the values in the following list. Only one flag may be chosen from those that affect horizontal and vertical alignment. In addition, only one of the two flags that alter the current position can be chosen:

Value	Meaning
TA_BASELINE	Specifies alignment of the point and the baseline of the chosen font.
TA_BOTTOM	Specifies alignment of the point and the bottom of the bounding rectangle.
TA_CENTER	Specifies alignment of the point and the horizontal center of the bounding rectangle.
TA_LEFT	Specifies alignment of the point and the left side of the bounding rectangle.
TA_NOUPDATECP	Specifies that the current position is not updated after each **TextOut** or **ExtTextOut** function call.
TA_RIGHT	Specifies alignment of the point and the right side of the bounding rectangle.

TA_TOP	Specifies alignment of the point and the top of the bounding rectangle.
TA_UPDATECP	Specifies that the current position is updated after each **TextOut** or **ExtTextOut** function call.

The defaults are TA_LEFT, TA_TOP, and TA_NOUPDATECP.

Return value The return value specifies the previous text alignment setting; the low-order word contains the horizontal alignment, and the high-order word contains the vertical alignment.

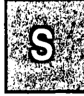
SetTextCharacterExtra

Syntax int SetTextCharacterExtra(hDC, nCharExtra)
function SetTextCharacterExtra(DC: HDC; CharExtra: Integer): Integer;

This function sets the amount of intercharacter spacing. GDI adds this spacing to each character, including break characters, when it writes a line of text to the device context.

Parameters

hDC	**HDC** Identifies the device context.
nCharExtra	**int** Specifies the amount of extra space (in logical units) to be added to each character. If the current mapping mode is not MM_TEXT, the *nCharExtra* parameter is transformed and rounded to the nearest pixel.



Return value The return value specifies the amount of the previous intercharacter spacing.

SetTextColor

Syntax DWORD SetTextColor(hDC, crColor)
function SetTextColor(DC: HDC; Color: TColorRef): Longint;

This function sets the text color to the color specified by the *crColor* parameter, or to the nearest physical color if the device cannot represent the color specified by *crColor*. GDI uses the text color to draw the face of each character written by the **TextOut** and **ExtTextOut** functions. GDI also uses the text color when converting bitmaps from color to monochrome and vice versa.

The background color for a character is specified by the **SetBkColor** and **SetBkMode** functions. For color-bitmap conversions, see the **BitBlt** and **StretchBlt** functions, earlier in this chapter.

Parameters

hDC	**HDC** Identifies the device context.
crColor	**COLORREF** Specifies the color of the text.

Return value The return value specifies an RGB color value for the previous text color.

SetTextJustification

Syntax int SetTextJustification(hDC, nBreakExtra, nBreakCount)
function SetTextJustification(DC: HDC; BreakExtra, BreakCount: Integer): Integer;

This function prepares GDI to justify a line of text using the justification parameters specified by the *nBreakExtra* and *nBreakCount* parameters. To justify text, GDI distributes extra pixels among break characters in a text line written by the **TextOut** function. The break character, used to delimit words, is usually the space character (ASCII 32), but may be defined by a font as some other character. The **GetTextMetrics** function can be used to retrieve a font's break character.

The **SetTextJustification** function prepares the justification by defining the amount of space to be added. The *nBreakExtra* parameter specifies the total amount of space (in logical units) to be added to the line. The *nBreakCount* parameter specifies how many break characters are in the line. The subsequent **TextOut** function distributes the extra space evenly between each break character in the line.

GetTextExtent is always used with the **SetTextJustification** function. The **GetTextExtent** function computes the width of a given line before justification. This width must be known before an appropriate *nBreakExtra* value can be computed.

SetTextJustification can be used to justify a line that contains multiple runs in different fonts. In this case, the line must be created piecemeal by justifying and writing each run separately.

Because rounding errors can occur during justification, GDI keeps a running error term that defines the current error. When justifying a line that contains multiple runs, **GetTextExtent** automatically uses this error term when it computes the extent of the next run, allowing **TextOut** to blend the error into the new run. After each line has been justified, this error term must be cleared to prevent it from being incorporated into the

next line. The term can be cleared by calling **SetTextJustification** with *nBreakExtra* set to zero.

Parameters

hDC — **HDC** Identifies the device context.

nBreakExtra — **int** Specifies the total extra space (in logical units) to be added to the line of text. If the current mapping mode is not MM_TEXT, the value identified by the *nBreakExtra* parameter is transformed and rounded to the nearest pixel.

nBreakCount — **int** Specifies the number of break characters in the line.

Return value The return value specifies the outcome of the function. It is 1 if the function is successful. Otherwise, it is zero.

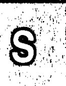
SetTimer

Syntax WORD SetTimer(hWnd, nIDEvent, wElapse, lpTimerFunc)
function SetTimer(Wnd: HWnd; IDEvent: Integer; Elapse: Word; TimerFunc: TFarProc): Word;

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

Parameters

hWnd — **HWND** Identifies the window to be associated with the timer. If *hWnd* is NULL, no window is associated with the timer.



nIDEvent — **int** Specifies a nonzero timer-event identifier if the *hWnd* parameter is not NULL.

wElapse — **WORD** Specifies the elapsed time (in milliseconds) between timer events.

lpTimerFunc — **FARPROC** Is the procedure-instance address of the function to be notified when the timer event takes place. If *lpTimerFunc* is NULL, the WM_TIMER message is placed in the application queue, and the **hwnd** member of the **MSG** structure contains the *hWnd* parameter given in the **SetTimer** function call. See the following "Comments" section for details.

Return value The return value specifies the integer identifier for the new timer event. If the *hWnd* parameter is NULL, an application passes this value to the

KillTimer function to kill the timer event. The return value is zero if the timer was not created.

Comments Timers are a limited global resource; therefore, it is important that an application check the value returned by the **SetTimer** function to verify that a timer is actually available.

To install a timer function, **SetTimer** must receive a procedure-instance address of the function, and the function must be exported in the application's module-definition file. A procedure-instance address can be created using the **MakeProcInstance** function.

The callback function must use the Pascal calling convention and must be declared **FAR**.

Callback function

WORD FAR PASCAL *TimerFunc(hWnd, wMsg, nIDEvent, dwTime)*
HWND *hWnd;*
WORD *wMsg;*
int *nIDEvent;*
DWORD *dwTime;*

TimerFunc is a placeholder for the application-supplied function name. The actual name must be exported by including it in an **EXPORTS** statement in the application's module-definition file.

Parameters

hWnd	Identifies the window associated with the timer event.
wMsg	Specifies the WM_TIMER message.
nIDEvent	Specifies the timer's ID.
dwTime	Specifies the current system time.

SetViewportExt

Syntax DWORD SetViewportExt(hDC, X, Y)
function SetViewportExt(DC: HDC; X, Y: Integer): Longint;

This function sets the *x*- and *y*-extents of the viewport of the specified device context. The viewport, along with the device-context window, defines how GDI maps points in the logical coordinate system to points in the coordinate system of the actual device. In other words, they define how GDI converts logical coordinates into device coordinates.

The *x*- and *y*-extents of the viewport define how much GDI must stretch or compress units in the logical coordinate system to fit units in the device coordinate system. For example, if the *x*-extent of the window is 2 and the *x*-extent of the viewport is 4, GDI maps two logical units (measured from the *x*-axis) into four device units. Similarly, if the *y*-extent of the window is 2 and the *y*-extent of the viewport is –1, GDI maps two logical units (measured from the *y*-axis) into one device unit.

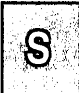
The extents also define the relative orientation of the *x*- and *y*-axes in both coordinate systems. If the signs of matching window and viewport extents are the same, the axes have the same orientation. If the signs are different, the orientation is reversed. For example, if the *y*-extent of the window is 2 and the *y*-extent of the viewport is –1, GDI maps the positive *y*-axis in the logical coordinate system to the negative *y*-axis in the device coordinate system. If the *x*-extents are 2 and 4, GDI maps the positive *x*-axis in the logical coordinate system to the positive *x*-axis in the device-coordinate system.

Parameters

hDC	**HDC** Identifies the device context.
X	**int** Specifies the *x*-extent of the viewport (in device units).
Y	**int** Specifies the *y*-extent of the viewport (in device units).

Return value The return value specifies the previous extents of the viewport. The previous *y*-extent is in the high-order word; the previous *x*-extent is in the low-order word. When an error occurs, the return value is zero.

Comments When the following mapping modes are set, calls to the **SetWindowExt** and **SetViewportExt** functions are ignored:



- MM_HIENGLISH
- MM_HIMETRIC
- MM_LOENGLISH
- MM_LOMETRIC
- MM_TEXT
- MM_TWIPS

When MM_ISOTROPIC mode is set, an application must call the **SetWindowExt** function before it calls **SetViewportExt**.

SetViewportOrg

Syntax DWORD SetViewportOrg(hDC, X, Y)
function SetViewportOrg(DC: HDC; X, Y: Integer): Longint;

This function sets the viewport origin of the specified device context. The viewport, along with the device-context window, defines how GDI maps points in the logical coordinate system to points in the coordinate system of the actual device. In other words, they define how GDI converts logical coordinates into device coordinates.

The viewport origin marks the point in the device coordinate system to which GDI maps the window origin, a point in the logical coordinate system specified by the **SetWindowOrg** function. GDI maps all other points by following the same process required to map the window origin to the viewport origin. For example, all points in a circle around the point at the window origin will be in a circle around the point at the viewport origin. Similarly, all points in a line that passes through the window origin will be in a line that passes through the viewport origin.

Parameters

hDC	**HDC** Identifies the device context.
X	**int** Specifies the *x*-coordinate (in device units) of the origin of the viewport. The value must be within the range of the device coordinate system.
Y	**int** Specifies the *y*-coordinate (in device units) of the origin of the viewport. The value must be within the range of the device coordinate system.

Return value The return value specifies the previous origin of the viewport (in device coordinates). The *y*-coordinate is in the high-order word; the *x*-coordinate is in the low-order word.

SetVoiceAccent

Syntax

int SetVoiceAccent(nVoice, nTempo, nVolume, nMode, nPitch)
function SetVoiceAccent(Voice, Tempo, Volume, Mode, Pitch: Integer): Integer;

This function places an accent (tempo, volume, mode, and pitch) in the voice queue specified by the *nVoice* parameter. The new accent replaces the previous accent and remains in effect until another accent is queued. An accent is not counted as a note.

An error occurs if there is insufficient room in the queue; the **SetVoiceAccent** function always leaves space for a single sync mark in the queue. If *nVoice* is out of range, the **SetVoiceAccent** function is ignored.

Parameters

nVoice	**int** Specifies a voice queue. The first voice queue is numbered 1.

nTempo	**int** Specifies the number of quarter notes played per minute. It can be any value from 32 to 255. The default is 120.
nVolume	**int** Specifies the volume level. It can be any value from 0 (lowest volume) to 255 (highest).
nMode	**int** Specifies how the notes are to be played. It can be any one of the following values:

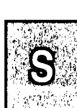
Value	Meaning
S_LEGATO	Note is held for the full duration and blended with the beginning of the next note.
S_NORMAL	Note is held for the full duration, coming to a full stop before the next note starts.
S_STACCATO	Note is held for only part of the duration, creating a pronounced stop between it and the next note.

nPitch	**int** Specifies the pitch of the notes to be played. It can be any value from 0 to 83. The pitch value is added to the note value, using modulo 84 arithmetic.

Return value The return value specifies the result of the function. It is zero if the function is successful. If an error occurs, the return value is one of the following values:

Parameters

S_SERDMD	Invalid mode
S_SERDTP	Invalid tempo
S_SERDVL	Invalid volume
S_SERQFUL	Queue full



SetVoiceEnvelope

Syntax int SetVoiceEnvelope(nVoice, nShape, nRepeat)
function SetVoiceEnvelope(Voice, Shape, RepeatCount: Integer): Integer;

This function queues the envelope (wave shape and repeat count) in the voice queue specified by the *nVoice* parameter. The new envelope replaces the previous one and remains in effect until the next **SetVoiceEnvelope** function call. An envelope is not counted as a note.

An error occurs if there is insufficient room in the queue; the **SetVoiceEnvelope** function always leaves space for a single sync mark in the queue. If *nVoice* is out of range, **SetVoiceEnvelope** is ignored.

Parameters *nVoice* **int** Specifies the voice queue to receive the envelope.

nShape	**int** Specifies an index to an OEM wave-shape table.
nRepeat	**int** Specifies the number of repetitions of the wave shape during the duration of one note.

Return value The return value specifies the result of the function. It is zero if the function is successful. If an error occurs, the return value is one of the following values:

Value	Meaning
S_SERDRC	Invalid repeat count
S_SERDSH	Invalid shape
S_SERQFUL	Queue full

SetVoiceNote

Syntax int SetVoiceNote(nVoice, nValue, nLength, nCdots)
function SetVoiceNote(Voice, Value, Length, Cdots: Integer): Integer;

This function queues a note that has the qualities given by the *nValue*, *nLength*, and *nCdots* parameters in the voice queue specified by the *nVoice* parameter. An error occurs if there is insufficient room in the queue. The function always leaves space in the queue for a single sync mark.

Parameters

nVoice	**int** Specifies the voice queue to receive the note. If *nVoice* is out of range, the **SetVoiceNote** function is ignored.
nValue	**int** Specifies 1 of 84 possible notes (seven octaves). If *nValue* is zero, a rest is assumed.
nLength	**int** Specifies the reciprocal of the duration of the note. For example, 1 specifies a whole note, 2 a half note, 4 a quarter note, and so on.
nCdots	**int** Specifies the duration of the note in dots. The duration is equal to *nLength* × (*nCdots* × 3/2).

Return value The return value specifies the result of the function. It is zero if the function is successful. If an error occurs, the return value is one of the following values:

Value	Meaning
S_SERDCC	Invalid dot count
S_SERDLN	Invalid note length
S_SERDNT	Invalid note
S_SERQFUL	Queue full

SetVoiceQueueSize

Syntax int SetVoiceQueueSize(nVoice, nBytes)
function SetVoiceQueueSize(Voice, Bytes: Integer): Integer;

This function allocates the number of bytes specified by the *nBytes* parameter for the voice queue specified by the *nVoice* parameter. If the queue size is not set, the default is 192 bytes, which is room for about 32 notes. All voice queues are locked in memory. The queues cannot be set while music is playing.

Parameters

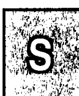
nVoice **int** Specifies a voice queue.

nBytes **int** Specifies the number of bytes in the voice queue.

Return value The return value specifies the result of the function. It is zero if the function is successful. If an error occurs, the return value is one of the following values:

Value	Meaning
S_SERMACT	Music active
S_SEROFM	Out of memory

SetVoiceSound



Syntax int SetVoiceSound(nVoice, lFrequency, nDuration)
function SetVoiceSound(Voice: Longint; Frequency: Longint; Duration: Integer): Integer;

This function queues the sound frequency and duration in the voice queue specified by the *nVoice* parameter.

Parameters

nVoice **int** Specifies a voice queue. The first voice queue is numbered 1.

lFrequency **long** Specifies the frequency. The high-order word contains the frequency in hertz, and the low-order word contains the fractional frequency.

nDuration **int** Specifies the duration of the sound (in clock ticks).

Return value The return value specifies the result of the function. It is zero if the function is successful. If an error occurs, the return value is one of the following values:

Value	Meaning
S_SERDDR	Invalid duration
S_SERDFQ	Invalid frequency
S_SERDVL	Invalid volume
S_SERQFUL	Queue full

SetVoiceThreshold

Syntax int SetVoiceThreshold(nVoice, nNotes)
function SetVoiceThreshold(Voice, Notes: Integer): Integer;

This function sets the threshold level for the given voice. When the number of notes remaining in the voice queue goes below that specified in the *nNotes* parameter, the threshold flag is set. If the queue level is below that specified in *nNotes* when the **SetVoiceThreshold** function is called, the flag is not set. The **GetThresholdStatus** function should be called to verify the current threshold status.

Parameters

nVoice **int** Specifies the voice queue to be set.

nNotes **int** Specifies the number of notes in the threshold level.

Return value The return value specifies the result of the function. It is zero if the function is successful. It is 1 if the number of notes specified in *nNotes* is out of range.

SetWindowExt

Syntax DWORD SetWindowExt(hDC, X, Y)
function SetWindowExt(DC: HDC; X, Y: Integer): Longint;

This function sets the *x*- and *y*-extents of the window associated with the specified device context. The window, along with the device-context viewport, defines how GDI maps points in the logical coordinate system to points in the device coordinate system.

The *x*- and *y*-extents of the window define how much GDI must stretch or compress units in the logical coordinate system to fit units in the device coordinate system. For example, if the *x*-extent of the window is 2 and the *x*-extent of the viewport is 4, GDI maps two logical units (measured from the *x*-axis) into four device units. Similarly, if the *y*-extent of the window is 2 and the *y*-extent of the viewport is –1, GDI maps two logical units (measured from the *y*-axis) into one device unit.

The extents also define the relative orientation of the *x*- and *y*-axes in both coordinate systems. If the signs of matching window and viewport extents are the same, the axes have the same orientation. If the signs are different, the orientation is reversed. For example, if the *y*-extent of the window is 2 and the *y*-extent of the viewport is –1, GDI maps the positive *y*-axis in the logical coordinate system to the negative *y*-axis in the device coordinate system. If the *x*-extents are 2 and 4, GDI maps the positive *x*-axis in the logical coordinate system to the positive *x*-axis in the device coordinate system.

Parameters

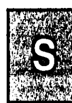
hDC	**HDC** Identifies the device context.
X	**int** Specifies the *x*-extent (in logical units) of the window.
Y	**int** Specifies the *y*-extent (in logical units) of the window.

Return value The return value specifies the previous extents of the window (in logical units). The *y*-extent is in the high-order word; the *x*-extent is in the low-order word. If an error occurs, the return value is zero.

Comments When the following mapping modes are set, calls to the **SetWindowExt** and **SetViewportExt** functions are ignored:

- MM_HIENGLISH
- MM_HIMETRIC
- MM_LOENGLISH
- MM_LOMETRIC
- MM_TEXT
- MM_TWIPS

When MM_ISOTROPIC mode is set, an application must call the **SetWindowExt** function before calling **SetViewportExt**.



SetWindowLong

Syntax LONG SetWindowLong(hWnd, nIndex, dwNewLong)
function SetWindowLong(Wnd: HWnd; Index: Integer; NewLong: Longint): Longint;

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

Parameters

hWnd	**HWND** Identifies the window.
nIndex	**int** Specifies the byte offset of the attribute to be changed. It may also be one of the following values:

Value	Meaning
GWL_EXSTYLE	Sets a new extended window style.
GWL_STYLE	Sets a new window style.
GWL_WNDPROC	Sets a new long pointer to the window procedure.

dwNewLong **DWORD** Specifies the replacement value.

Return value The return value specifies the previous value of the specified long integer.

Comments If the **SetWindowLong** function and the GWL_WNDPROC index are used to set a new window function, that function must have the window-function form and be exported in the module-definition file of the application. For more information, see the **RegisterClass** function, earlier in this chapter.

Calling **SetWindowLong** with the GCL_WNDPROC index creates a subclass of the window class used to create the window. See Chapter 1, "Window manager interface functions," for more information on window subclassing. An application should not attempt to create a window subclass for standard Windows controls such as combo boxes and buttons.

To access any extra four-byte values allocated when the window-class structure was created, use a positive byte offset as the index specified by the *nIndex* parameter, starting at zero for the first four-byte value in the extra space, 4 for the next four-byte value and so on.

SetWindowOrg

Syntax DWORD SetWindowOrg(hDC, X, Y)
function SetWindowOrg(DC: HDC; X, Y: Integer): Longint;

This function sets the window origin of the specified device context. The window, along with the device-context viewport, defines how GDI maps points in the logical coordinate system to points in the device coordinate system.

The window origin marks the point in the logical coordinate system from which GDI maps the viewport origin, a point in the device coordinate system specified by the **SetWindowOrg** function. GDI maps all other points by following the same process required to map the window origin to the viewport origin. For example, all points in a circle around the point at the window origin will be in a circle around the point at the viewport

origin. Similarly, all points in a line that passes through the window origin will be in a line that passes through the viewport origin.

Parameters

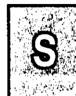
hDC	**HDC** Identifies the device context.
X	**int** Specifies the logical *x*-coordinate of the new origin of the window.
Y	**int** Specifies the logical *y*-coordinate of the new origin of the window.

Return value The return value specifies the previous origin of the window. The previous *y*-coordinate is in the high-order word; the previous *x*-coordinate is in the low-order word.

SetWindowPos

Syntax void SetWindowPos(hWnd, hWndInsertAfter, X, Y, cx, cy, wFlags)
procedure SetWindowPos(Wnd, WndInsertAfter: HWnd; X, Y, cx, cy: Integer; Flags: Word)

This function changes the size, position, and ordering of child, pop-up, and top-level windows. Child, pop-up, and top-level windows are ordered according to their appearance on the screen; the window above all other windows receives the highest rank, and it is the first window in the list. This ordering is recorded in a window list.



Parameters

hWnd	**HWND** Identifies the window that will be positioned.
hWndInsertAfter	**HWND** Identifies a window in the window-manager's list that will precede the window identified by the *hWnd* parameter. If the window identified by the *hWnd* parameter has the WS_ES_TOPMOST style and *hWndInsertAfter* is –1, the window is placed at the top of the hierarchy of topmost windows and remains above all non-topmost windows, even when inactive. If the window has the WS_ES_TOPMOST style and *hWndInsertAfter* is 1, the window is no longer treated as a topmost window and is placed below all other windows.
X	**int** Specifies the *x*-coordinate of the window's upper-left corner.
Y	**int** Specifies the *y*-coordinate of the window's upper-left corner.

cx	**int** Specifies the new window's width.
cy	**int** Specifies the new window's height.
wFlags	**WORD** Specifies one of eight possible 16-bit values that affect the sizing and positioning of the window. It must be one of the following values:

Value	Meaning
SWP_DRAWFRAME	Draws a frame (defined in the window's class description) around the window.
SWP_HIDEWINDOW	Hides the window.
SWP_NOACTIVATE	Does not activate the window.
SWP_NOMOVE	Retains current position (ignores the *x* and *y* parameters).
SWP_NOSIZE	Retains current size (ignores the *cx* and *cy* parameters).
SWP_NOREDRAW	Does not redraw changes.
SWP_NOZORDER	Retains current ordering (ignores the *hWndInsertAfter* parameter).
SWP_SHOWWINDOW	Displays the window.

Return value None.

Comments If the SWP_NOZORDER flag is not specified, Windows places the window identified by the *hWnd* parameter in the position following the window identified by the *hWndInsertAfter* parameter. If *hWndInsertAfter* is NULL, Windows places the window identified by *hWnd* at the top of the list. If *hWndInsertAfter* is set to 1, Windows places the window identified by *hWnd* at the bottom of the list.

If the SWP_SHOWWINDOW or the SWP_HIDEWINDOW flags are set, scrolling and moving cannot be done simultaneously.

All coordinates for child windows are relative to the upper-left corner of the parent window's client area.

SetWindowsHook

Syntax FARPROC SetWindowsHook(nFilterType, lpFilterFunc)
function SetWindowsHook(FilterType: Integer; FilterFunc: TFarProc): TFarProc;

This function installs a filter function in a chain. A filter function processes events before they are sent to an application's message loop in the WinMain function. A chain is a linked list of filter functions of the same type.

Parameters

nFilterType **int** Specifies the system hook to be installed. It can be any one of the following values:

Value	Meaning
WH_CALLWNDPROC	Installs a window-function filter.
WH_GETMESSAGE	Installs a message filter.
WH_JOURNALPLAYBACK	Installs a journaling playback filter.
WH_JOURNALRECORD	Installs a journaling record filter.
WH_KEYBOARD	Installs a keyboard filter.
WH_MSGFILTER	Installs a message filter.
WH_SYSMSGFILTER	Installs a system-wide message filter.

lpFilterFunc **FARPROC** Is the procedure-instance address of the filter function to be installed. See the following "Comments" section for details.

Return value

The return value points to the procedure-instance address of the previously installed filter (if any). It is NULL if there is no previous filter. The application or library that calls the **SetWindowsHook** function should save this return value in the library's data segment. The fourth argument of the **DefHookProc** function points to the location in memory where the library saves this return value.

The return value is –1 if the function fails.

Comments

The WH_CALLWNDPROC hook will affect system performance. It is supplied for debugging purposes only.

The system hooks are a shared resource. Installing a hook affects all applications. Most hook functions must be in libraries. The only exception is WH_MSGFILTER, which is task-specific. System hooks should be restricted to special-purpose applications or as a development aid during debugging of an application. Libraries that no longer need the hook should remove the filter function.

To install a filter function, the **SetWindowsHook** function must receive a procedure-instance address of the function, and the function must be exported in the library's module-definition file. Libraries can pass the procedure address directly. Tasks must use **MakeProcInstance** to get a

procedure-instance address. Dynamic-link libraries must use **GetProcAddress** to get a procedure-instance address.

The following section describes how to support the individual hook functions.

WH_CALLWNDPROC

Windows calls the WH_CALLWNDPROC filter function whenever the **SendMessage** function is called. Windows does not call the filter function when the **PostMessage** function is called.

The filter function must use the Pascal calling convention and must be declared **FAR**. The filter function must have the following form:

Filter Function

DWORD FAR PASCAL *FilterFunc(nCode, wParam, lParam)*
int *nCode;*
WORD *wParam;*
DWORD *lParam;*

FilterFunc is a placeholder for the application- or library-supplied function name. The actual name must be exported by including it in an **EXPORTS** statement in the library's module-definition file.

Parameters

nCode Specifies whether the filter function should process the message or call the **DefHookProc** function. If the *nCode* parameter is less than zero, the filter function must pass the message to **DefHookProc** without further processing and return the value returned by **DefHookProc**.

wParam Specifies whether the message is sent by the current task. It is nonzero if the message is sent; otherwise, it is NULL.

lParam Points to a data structure that contains details about the message intercepted by the filter. The following shows the order, type, and description of each field of the data structure:

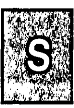
Field	Type/Description
hlParam	**WORD** Contains the high-order word of the *lParam* parameter of the message received by the filter.
llParam	**WORD** Contains the low-order word of the *lParam*

	parameter of the message received by the filter.
wParam	**WORD** Contains the *wParam* parameter of the message received by the filter.
wMsg	**WORD** Contains the message received by the filter.
hWnd	**WORD** Contains the window handle of the window that is to receive the message.

Comments The WH_CALLWNDPROC filter function can examine or modify the message as desired. Once it returns control to Windows, the message, with any modifications, is passed on to the window function. The filter function does not require a return value.

WH_GETMESSAGE

Windows calls the WH_GETMESSAGE filter function whenever the **GetMessage** function is called. Windows calls the filter function immediately after **GetMessage** has retrieved a message from an application queue. The filter function must use the Pascal calling convention and must be declared **FAR**. The filter function must have the following form:



Filter Function

DWORD FAR PASCAL *FilterFunc(nCode, wParam, lParam)*
int *nCode;*
WORD *wParam;*
DWORD *lParam;*

FilterFunc is a placeholder for the library-supplied function name. The actual name must be exported by including it in an **EXPORTS** statement in the library's module-definition file.

Parameters

nCode	Specifies whether the filter function should process the message or call the **DefHookProc** function. If the *nCode* parameter is less than zero, the filter function must pass the message to **DefHookProc** without further processing and return the value returned by **DefHookProc**.

wParam	Specifies a NULL value.
lParam	Points to a message structure.

Comments The WH_GETMESSAGE filter function can examine or modify the message as desired. Once it returns control to Windows, the **GetMessage** function returns the message, with any modifications, to the application that originally called it. The filter function does not require a return value.

WH_JOURNALPLAYBACK

Windows calls the WH_JOURNALPLAYBACK filter function whenever a request for an event message is made. The function is intended to be used to supply a previously recorded event message.

The filter function must use the Pascal calling convention and must be declared **FAR**. The filter function must have the following form:

Filter Function

DWORD FAR PASCAL *FilterFunc*(*nCode*, *wParam*, *lParam*);
int *nCode*;
WORD *wParam*;
DWORD *lParam*;

FilterFunc is a placeholder for the library-supplied function name. The actual name must be exported by including it in an **EXPORTS** statement in the library's module-definition file.

Parameters

nCode	Specifies whether the filter function should process the message or call the **DefHookProc** function. If the *nCode* parameter is less than zero, the filter function must pass the message to **DefHookProc** without further processing and return the value returned by **DefHookProc**.
wParam	Specifies a NULL value.
lParam	Points to the message being processed by the filter function.

Comments The WH_JOURNALPLAYBACK function should copy an event message to the *lParam* parameter. The message must have been previously recorded by using the WH_JOURNALRECORD filter. It should not modify the message. The return value should be the amount of time (in clock ticks) Windows should wait before processing the message. This time can be computed by calculating the difference between the **time** fields in the current and previous event messages. If the function returns

zero, the message is processed immediately. Once it returns control to Windows, the message continues to be processed. If the *nCode* parameter is HC_SKIP, the filter function should prepare to return the next recorded event message on its next call.

While the WH_JOURNALPLAYBACK function is in effect, Windows ignores all mouse and keyboard input.

WH_JOURNALRECORD

Windows calls the WH_JOURNALRECORD filter function whenever it processes a message from the event queue. The filter can be used to record the event for later playback.

The filter function must use the Pascal calling convention and must be declared **FAR**. The filter function must have the following form:

Filter Function

DWORD FAR PASCAL *FilterFunc*(*nCode*, *wParam*, *lParam*);
int *nCode*;
WORD *wParam*;
DWORD *lParam*;

FilterFunc is a placeholder for the library-supplied function name. The actual name must be exported by including it in an **EXPORTS** statement in the library's module-definition file.

Parameters

nCode	Specifies whether the filter function should process the message or call the **DefHookProc** function. If the *nCode* parameter is less than zero, the filter function must pass the message to **DefHookProc** without further processing and return the value returned by **DefHookProc**.
wParam	Specifies a NULL value.
lParam	Points to a message structure.

Comments

The WH_JOURNALRECORD function should save a copy of the event message for later playback. It should not modify the message. Once it returns control to Windows, the message continues to be processed. The filter function does not require a return value.

WH_KEYBOARD

Windows calls the WH_KEYBOARD filter function whenever the application calls the **GetMessage** or **PeekMessage** function and there is a keyboard event (WM_KEYUP or WM_KEYDOWN) to process.

The filter function must use the Pascal calling convention and must be declared **FAR**. The filter function must have the following form:

Filter Function

DWORD FAR PASCAL *FilterFunc*(*nCode*, *wParam*, *lParam*)
int *nCode*;
WORD *wParam*;
DWORD *lParam*;

FilterFunc is a placeholder for the library-supplied function name. The actual name must be exported by including it in an **EXPORTS** statement in the library's module-definition file.

Parameters

nCode — Specifies whether the filter function should process the message or call the **DefHookProc** function. If this value is HC_NOREMOVE, the application is using the **PeekMessage** function with the PM_NOREMOVE option and the message will not be removed from the system queue. If the *nCode* parameter is less than zero, the filter function must pass the message to **DefHookProc** without further processing and return the value returned by **DefHookProc**.

wParam — Specifies the virtual-key code of the given key.

lParam — Specifies the repeat count, scan code, key-transition code, previous key state, and context code, as shown in the following list. Bit 1 is the low-order bit:

Bit	Value
0–15 (low-order word)	Repeat count (the number of times the keystroke is repeated as a result of the user holding down the key).
16–23 (low byte of high-order word)	Scan code (OEM-dependent value).
24[1]	Extended key (1 if it is an extended key).
25–26	Not used.
27–28	Used internally by Windows.
30	Previous key state (1 if the key was held down before the message was sent, 0 if the key was up).

31	Transition state (1 if the key is being released, 0 if the key is being pressed).

[1] (Context code (1 if the ALT key was held down while the key was pressed, 0 otherwise)

Return value The return value specifies what should happen to the message. It is zero if the message should be processed by Windows; it is 1 if the message should be discarded.

WH_MSGFILTER

Windows calls the WH_MSGFILTER filter function whenever a dialog box, message box, or menu has retrieved a message, and before it has processed that message. The filter allows an application to process or modify the messages.

This is the only task-specific filter. A task may install this filter.

The WH_MSGFILTER filter function must use the Pascal calling convention and must be declared **FAR**. The filter function must have the following form:

Filter Function

DWORD FAR PASCAL *FilterFunc(nCode, wParam, lParam*)
int *nCode;*
WORD *wParam;*
DWORD *lParam;*

FilterFunc is a placeholder for the library- or application-supplied function name. The actual name must be exported by including it in an **EXPORTS** statement in the application's module-definition file.

Parameters

nCode Specifies the type of message being processed. It must be one of the following values:

Value	**Meaning**
MSGF_DIALOGBOX	Processing messages inside a dialog-box or message-box function.
MSGF_MENU	Processing keyboard and mouse messages in a menu.

If the *nCode* parameter is less than zero, the filter function must pass the message to **DefHookProc** without further processing and return the value returned by **DefHookProc**.

wParam Specifies a NULL value.

lParam Points to the message structure.

Return value The return value specifies the outcome of the function. It is nonzero if the hook function processes the message. Otherwise, it is zero.

WH_SYSMSGFILTER

Windows calls the WH_SYSMSGFILTER filter function whenever a dialog box, message box, or menu has retrieved a message and before it has processed that message. The filter allows an application to process or modify messages for any application in the system.

The filter function must use the Pascal calling convention and must be declared **FAR**. The filter function must have the following form:

Filter Function

DWORD FAR PASCAL *FilterFunc(nCode, wParam, lParam)*
int *nCode;*
WORD *wParam;*
DWORD *lParam;*

FilterFunc is a placeholder for the library-supplied function name. The actual name must be exported by including it in an **EXPORTS** statement in the library's module-definition file.

Parameters

nCode Specifies the type of message being processed. It must be one of the following values:

Value	Meaning
MSGF_DIALOGBOX	Processing messages inside the **DialogBox** function.
MSGF_MENU	Processing keyboard and mouse messages in menu.
MSGF_MESSAGEBOX	Processing messages inside the **MessageBox** function.

If the *nCode* parameter is less than zero, the filter function must pass the message to **DefHookProc** without further processing and return the value returned by **DefHookProc**.

wParam Specifies a NULL value.

lParam Points to the message structure.

Return value The return value specifies the outcome of the function. It is nonzero if the hook function processes the message. Otherwise, it is zero.

SetWindowText

Syntax void SetWindowText(hWnd, lpString)
procedure SetWindowText(Wnd: HWnd; Str: PChar);

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

Parameters

hWnd **HWND** Identifies the window or control whose text is to be changed.

lpString **LPSTR** Points to a null-terminated character string.

Return value None.

SetWindowWord

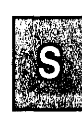
Syntax WORD SetWindowWord(hWnd, nIndex, wNewWord)
function SetWindowWord(Wnd: HWnd; Index: Integer; NewWord: Word): Word;

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

Parameters

hWnd **HWND** Identifies the window to be modified.

nIndex **int** Specifies the byte offset of the word to be changed. It can also be one of the following values:



Value	**Meaning**
GWW_HINSTANCE	Instance handle of the module that owns the window.
GWW_ID	Control ID of the child window.

wNewWord **WORD** Specifies the replacement value.

Return value The return value specifies the previous value of the specified word.

Comments To access any extra two-byte values allocated when the window-class structure was created, use a positive byte offset as the index specified by the *nIndex* parameter, starting at zero for the first two-byte value in the extra space, 2 for the next two-byte value and so on.

ShowCaret

Syntax void ShowCaret(hWnd)
procedure ShowCaret(Wnd: HWnd);

This function shows the caret on the display at the caret's current position. Once shown, the caret begins flashing automatically.

The **ShowCaret** function shows the caret only if it has a current shape and has not been hidden two or more times in a row. If the caret is not owned by the given window, the caret is not shown. If the *hWnd* parameter is NULL, the **SetCaret** function shows the caret only if it is owned by a window in the current task.

Hiding the caret is accumulative. If the **HideCaret** function has been called five times in a row, **ShowCaret** must be called five times to show the caret.

Parameters *hWnd* **HWND** Identifies the window that owns the caret, or is NULL to specify indirectly the owner window in the current task.

Return value None.

Comments The caret is a shared resource. A window should show the caret only when it has the input focus or is active.

ShowCursor

Syntax int ShowCursor(bShow)
function ShowCursor(Show: Bool): Integer;

This function shows or hides the cursor. The **ShowCursor** function actually sets an internal display counter that determines whether the cursor should be displayed. If the *bShow* parameter is nonzero, **ShowCursor** adds one to the display count. If *bShow* is zero, the display count is decreased by one. The cursor is displayed only if the display count is greater than or equal to zero. Initially, the display count is zero if a mouse is installed. Otherwise, it is –1.

Parameters *bShow* **BOOL** Specifies whether the display count is to be increased or decreased. The display count is increased if *bShow* is nonzero. Otherwise, it is decreased.

Return value The return value specifies the new display count.

Comments The cursor is a shared resource. A window that hides the cursor should show the cursor before the cursor leaves its client area, or before the window relinquishes control to another window.

ShowOwnedPopups

Syntax void ShowOwnedPopups(hWnd, fShow)
procedure ShowOwnedPopups(Wnd: HWnd; Show: Bool);

This function shows or hides all pop-up windows that are associated with the *hWnd* parameter. If the *fShow* parameter is nonzero, all hidden pop-up windows are shown; if *fShow* is zero, all visible pop-up windows are hidden.

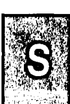
Parameters

hWnd — **HWND** Identifies the window that owns the pop-up windows that are to be shown or hidden.

fShow — **BOOL** Specifies whether or not pop-up windows are hidden. It is nonzero if all hidden pop-up windows should be shown; it is zero if all visible pop-up windows should be hidden.

Return value None.

ShowScrollBar



Syntax void ShowScrollBar(hWnd, wBar, bShow)
procedure ShowScrollBar(Wnd: HWnd; Bar: Word; Show: Bool);

This function displays or hides a scroll bar, depending on the value of the *bShow* parameter. If *bShow* is nonzero, the scroll bar is displayed; if *bShow* is zero, the scroll bar is hidden.

Parameters

hWnd — **HWND** Identifies a window that contains a scroll bar in its nonclient area if the *wBar* parameter is SB_HORZ, SB_VERT, or SB_BOTH. If *wBar* is SB_CTL, *hWnd* identifies a scroll-bar control.

wBar — **WORD** Specifies whether the scroll bar is a control or part of a window's nonclient area. If it is part of the nonclient area, *wBar* also indicates whether the scroll bar is positioned horizontally, vertically, or both. It must be one of the following values:

Value	Meaning
SB_BOTH	Specifies the window's horizontal and vertical scroll bars.
SB_CTL	Specifies that the scroll bar is a control.
SB_HORZ	Specifies the window's horizontal scroll bar.
SB_VERT	Specifies the window's vertical scroll bar.

bShow **BOOL** Specifies whether or not Windows hides the scroll bar. If *bShow* is zero, the scroll bar is hidden. Otherwise, the scroll bar is displayed.

Return value None.

Comments An application should not call this function to hide a scroll bar while processing a scroll-bar notification message.

ShowWindow

Syntax BOOL ShowWindow(hWnd, nCmdShow)
function ShowWindow(Wnd: HWnd; CmdShow: Integer): Bool;

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

Parameters

hWnd **HWND** Identifies the window.

nCmdShow **int** Specifies how the window is to be shown. It must be one of the values shown in Table 4.18, "Window states."

Return value The return value specifies the previous state of the window. It is nonzero if the window was previously visible. It is zero if the window was previously hidden.

Comments The **ShowWindow** function must be called only once per program with the *nCmdShow* parameter from the WinMain function. Subsequent calls to **ShowWindow** must use one of the values listed above, instead of one specified by the *nCmdShow* parameter from the WinMain function. Table 4.18 lists the values for the *nCmdShow* parameter:

Table 4.18 Window states

Value	Meaning
SW_HIDE	Hides the window and passes activation to another window.
SW_MINIMIZE	Minimizes the specified window and activates the top-level window in the window-manager's list.
SW_RESTORE	Same as SW_SHOWNORMAL.

Table 4.18: Window states (continued)

SW_SHOW	Activates a window and displays it in its current size and position.
SW_SHOWMAXIMIZED	Activates the window and displays it as a maximized window.
SW_SHOWMINIMIZED	Activates the window and displays it as iconic.
SW_SHOWMINNOACTIVE	Displays the window as iconic. The window that is currently active remains active.
SW_SHOWNA	Displays the window in its current state. The window that is currently active remains active.
SW_SHOWNOACTIVATE	Displays a window in its most recent size and position. The window that is currently active remains active.
SW_SHOWNORMAL	Activates and displays a window. If the window is minimized or maximized, Windows restores it to its original size and position.

SizeofResource

Syntax WORD SizeofResource(hInstance, hResInfo)
function SizeofResource(Instance, ResInfo: THandle): Word;

This function supplies the size (in bytes) of the specified resource. It is typically used with the **AccessResource** function to prepare memory to receive a resource from the file.

Parameters

hInstance **HANDLE** Identifies the instance of the module whose executable file contains the resource.

hResInfo **HANDLE** Identifies the desired resource. This handle is assumed to have been created by using the **FindResource** function.

Return value The return value specifies the number of bytes in the resource. It is zero if the resource cannot be found.

Comments The value returned may be larger than the actual resource due to alignment. An application should not rely upon this value for the exact size of a resource.

StartSound

Syntax intStartSound( )
function StartSound: Integer;

S

This function starts play in each voice queue. It is not destructive, so it may be called any number of times to replay the current queues.

Parameters None.

Return value Although the return-value type is integer, its contents should be ignored.

StopSound

Syntax int StopSound( )
function StopSound: Integer;

This function stops playing all voice queues, then flushes the contents of the queues. The sound driver for each voice is turned off.

Parameters None.

Return value Although the return-value type is integer, its contents should be ignored.

StretchBlt

Syntax BOOLStretchBlt(hDestDC, X, Y, nWidth, nHeight, hSrcDC, XSrc, YSrc, nSrcWidth, nSrcHeight, dwRop)
function StretchBlt(DestDC: HDC; X, Y, Width, Height: Integer; SrcDC: HDC; XSrc, YSrc, SrcWidth, SrcHeight: Integer; Rop: Longint): Bool;

__StretchBlt__ creates a mirror image of a bitmap if the signs of the nSrcWidth and nWidth or nSrcHeight and nHeight parameters differ. If nSrcWidth and nWidth have different signs, it creates a mirror image of the bitmap along the x-axis. If nSrcHeight and nHeight have different signs, it creates a mirror image of the bitmap along the y-axis.

This function moves a bitmap from a source into a destination rectangle, stretching or compressing the bitmap if necessary to fit the dimensions of the destination rectangle. The **StretchBlt** function uses the stretching mode of the destination device context (set by the **SetStretchBltMode** function) to determine how to stretch or compress the bitmap. **StretchBlt** moves the bitmap from the source device given by the *hSrcDC* parameter to the destination device given by the *hDestDC* parameter. The *XSrc*, *YSrc*, *nSrcWidth*, and *nSrcHeight* parameters define the origin and dimensions of the source rectangle. The *X*, *Y*, *nWidth*, and *nHeight* parameters give the origin and dimensions of the destination rectangle. The raster operation specified by the *dwRop* parameter defines how the source bitmap and the bits already on the destination device are combined.

Parameters

hDestDC	**HDC** Identifies the device context to receive the bitmap.
X	**int** Specifies the logical *x*-coordinate of the upper-left corner of the destination rectangle.
Y	**int** Specifies the logical *y*-coordinate of the upper-left corner of the destination rectangle.

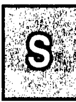
nWidth	**int** Specifies width (in logical units) of destination rectangle.
nHeight	**int** Specifies height (in logical units) of destination rectangle.
hSrcDC	**HDC** Identifies device context containing source bitmap.
XSrc	**int** Specifies the logical *x*-coordinate of the upper-left corner of the source rectangle.
YSrc	**int** Specifies the logical *y*-coordinate of the upper-left corner of the source rectangle.
nSrcWidth	**int** Specifies width (logical units) of source rectangle.
nSrcHeight	**int** Specifies height (logical units) of source rectangle.
dwRop	**DWORD** Specifies the raster operation to be performed. Raster operation codes define how GDI combines colors in output operations that involve a current brush, a possible source bitmap, and a destination bitmap. For a list of raster-operation codes, see the **BitBlt** function.

Return value The return value specifies whether the bitmap is drawn. It is nonzero if the bitmap is drawn. Otherwise, it is zero.

Comments **StretchBlt** stretches or compresses the source bitmap in memory, then copies the result to the destination. If a pattern is to be merged with the result, it is not merged until the stretched source bitmap is copied to the destination.

If a brush is used, it is the selected brush in the destination device context.

The destination coordinates are transformed according to the destination device context; the source coordinates are transformed according to the source device context.



If destination, source, and pattern bitmaps do not have the same color format, **StretchBlt** converts the source and pattern bitmaps to match the destination bitmaps. The foreground and background colors of the destination are used in the conversion.

If **StretchBlt** must convert a monochrome bitmap to color, it sets white bits (1) to background color and black bits (0) to foreground color. To convert color to monochrome, it sets pixels that match the background color to white (1), and sets all other pixels to black (0). The foreground and background colors of the device context with color are used.

Not all devices support the **StretchBlt** function. For more information, see the RC_BITBLT capability in the **GetDeviceCaps** function, earlier in this chapter.

StretchDIBits 3.0

Syntax WORD StretchDIBits(hDC, DestX, DestY, wDestWidth, wDestHeight, SrcX, SrcY, wSrcWidth, wSrcHeight, lpBits, lpBitsInfo, wUsage, dwRop)
function StretchDIBits(DC: HDC; DestX, DestY, DestWidth, DestHeight, SrcX, SrcY, SrcWidth, SrcHeight: Word; Bits: Pointer; var BitsInfo: TBitmapInfo; Usage: Word; Rop: Longint): Integer;

This function moves a device-independent bitmap (DIB) from a source rectangle into a destination rectangle, stretching or compressing the bitmap if necessary to fit the dimensions of the destination rectangle. The **StretchDIBits** function uses the stretching mode of the destination device context (set by the **SetStretchBltMode** function) to determine how to stretch or compress the bitmap.

StretchDIBits moves the bitmap from the device-independent bitmap specified by the *lpBits*, *lpBitsInfo*, and *wUsage* parameters to the destination device specified by the *hDC* parameter. The *XSrc*, *YSrc*, *wSrcWidth*, and *wSrcHeight* parameters define the origin and dimensions of the source rectangle. The origin of coordinate system of the device-independent bitmap is the lower-left corner. The *DestX*, *DestY*, *wDestWidth*, and *wDestHeight* parameters give the origin and dimensions of the destination rectangle. The origin of the coordinates of the destination depends on the current mapping mode of the device context. See the **SetMapMode** function earlier in this chapter for more information on mapping modes.

The raster operation specified by the *dwRop* parameter defines how the source bitmap and the bits already on the destination device are combined.

StretchDIBits creates a mirror image of a bitmap if the signs of the *wSrcWidth* and *wDestWidth* or *wSrcHeight* and *wDestHeight* parameters differ. If *wSrcWidth* and *nWidth* have different signs, the function creates a mirror image of the bitmap along the *x*-axis. If *wSrcHeight* and *nHeight* have different signs, the function creates a mirror image of the bitmap along the *y*-axis.

Parameters

hDC	**HDC** Identifies the destination device context for a display surface or memory bitmap.
DestX	**WORD** Specifies the *x*-coordinate (in logical units) of the origin of the destination rectangle.
DestY	**WORD** Specifies the *y*-coordinate (in logical units) of the origin of the destination rectangle.

wDestWidth	**WORD** Specifies the *x*-extent (in logical units) of the destination rectangle.
wDestHeight	**WORD** Specifies the *y*-extent (in logical units) of the destination rectangle.
SrcX	**WORD** Specifies the *x*-coordinate (in pixels) of the source in the DIB.
SrcY	**WORD** Specifies the *y*-coordinate (in pixels) of the source in the DIB.
wSrcWidth	**WORD** Specifies the width (in pixels) of the source rectangle in the DIB.
wSrcHeight	**WORD** Specifies the height (in pixels) of the source rectangle in the DIB.
lpBits	**LPSTR** Points to the DIB bits that are stored as an array of bytes.
lpBitsInfo	**LPBITMAPINFO** Points to a **BITMAPINFO** data structure that contains information about the DIB.
wUsage	**WORD** Specifies whether the **bmiColors[ ]** fields of the *lpBitsInfo* parameter contain explicit RGB values or indexes into the currently realized logical palette. The *wUsage* parameter must be one of the following values:

Value	**Meaning**
DIB_PAL_COLORS	The color table consists of an array of 16-bit indexes into the currently realized logical palette.
DIB_RGB_COLORS	The color table contains literal RGB values.

dwRop	**DWORD** Specifies the raster operation to be performed. Raster operation codes define how GDI combines colors in output operations that involve a current brush, a possible source bitmap, and a destination bitmap. For a list of raster-operation codes, see the **BitBlt** function, earlier in this chapter. For a complete list of the operations, see Chapter 11, "Binary and ternary raster-operation codes," in *Reference, Volume 2*.

Return value The return value is the number of scan lines copied.

S

Comments This function also accepts a device-independent bitmap specification formatted for Microsoft OS/2 Presentation Manager versions 1.1 and 1.2 if the *lpBitsInfo* parameter points to a **BITMAPCOREINFO** data structure.

SwapMouseButton

Syntax BOOL SwapMouseButton(bSwap)
function SwapMouseButton(Swap: Bool): Bool;

This function reverses the meaning of left and right mouse buttons. If the *bSwap* parameter is TRUE, the left button generates right-button mouse messages and the right button generates left-button messages. If *bSwap* is FALSE, the buttons are restored to their original meaning.

Parameters *bSwap* **BOOL** Specifies whether the button meanings are reversed or restored.

Return value The return value specifies the outcome of the function. It is TRUE if the fuction reversed the meaning of the mouse buttons. Otherwise, it is FALSE.

Comments Button swapping is provided as a convenience to people who use the mouse with their left hands. The **SwapMouseButton** function is usually called by the control panel only. Although applications are free to call the function, the mouse is a shared resource and reversing the meaning of the mouse button affects all applications.

SwapRecording 3.0

Syntax void SwapRecording(wFlag)
procedure SwapRecording(Flag: Word);

When running Microsoft Windows Swap, this function begins or ends analyzing swapping behavior. For more information on Swap, see *Tools*.

Parameters *wFlag* **WORD** Specifies whether Swap is to start or stop analyzing swapping behavior. The following are acceptable values:

Value	Meaning
0	Specifies that Swap stop analyzing.
1	Record swap calls, discard swap returns.
2	Same as 1, plus calls through thunks. This option records a large amount of data.

Return value None.

SwitchStackBack 3.0

Syntax void SwitchStackBack( )
procedure SwitchStackBack;

This function returns the stack of the current task to the task's data segment after it had been previously redirected by the **SwitchTasksBack** function.

Parameters None.

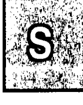
Return value None.

Comments This function preserves the contents of the AX:DX register when it returns.

SwitchStackTo 3.0

Syntax void SwitchStackTo(wStackSegment, wStackPointer, wStackTop)
procedure SwitchStackTo(StackSegment, StackPointer, StackTop: Word);

This function changes the stack of the current task to the segment identified by the *wStackSegment* parameter.



Dynamic-link libraries (DLLs) do not have a stack; instead, a DLL uses the stack of the task which calls the library. As a result, DLL functions that assume that the contents of the code-segment (CS) and stack-segment (SS) registers are the same will fail. The **SwitchStackTo** function redirects the stack of the task to the data segment of a DLL, permitting the DLL to call these functions. **SwitchStackTo** copies the arguments on the stack of the task to the new stack location.

Parameters

wStackSegment	**WORD** Specifies the data segment which is to contain the stack.
wStackPointer	**WORD** Specifies the offset of the beginning of the stack in the data segment.
wStackTop	**WORD** Specifies the offset of the top of the stack from the beginning of the stack.

Return value None.

Comments A task can call **SwitchStackTo** before calling a function in a DLL that assumes the CS and DS registeres are equal. When the DLL function returns, the task must then call **SwitchStackBack** to redirect its stack to its own data segment.

A DLL can also call **SwitchStackTo** before calling a routine that assumes CS and DS are equal and then call **SwitchStackBack** before returning to the task that called the DLL function.

Calls to **SwitchStackTo** and **SwitchStackBack** cannot be nested. That is, after calling **SwitchStackTo**, a program must call **SwitchStackBack** before calling **SwitchStackTo** again.

SyncAllVoices

Syntax int SyncAllVoices( )
function SyncAllVoices: Integer;

This function queues a sync mark in each queue. Upon encountering a sync mark in a voice queue, the voice is turned off until sync marks are encountered in all other queues. This forces synchronization among all voices.

Parameters None.

Return value The return value specifies the result of the function. It is zero if the function is successful. If a voice queue is full, the return value is S_SERQFUL.

TabbedTextOut 3.0

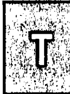
Syntax

long TabbedTextOut(hDC, X, Y lpString, nCount, nTabPositions, lpnTabStopPositions, nTabOrigin)
function TabbedTextOut(DC: HDC; X, Y: Integer; Str: PChar; Count, TabPositions: Integer; var TabStopPositions; TabOrigin: Integer): Longint;

This function writes a character string on the specified display, using the currently selected font and expanding tabs to the columns specified in the *lpnTabStopPositions* field.

Parameters

hDC	**HDC** Identifies the device context.
X	**int** Specifies the logical *x*-coordinate of the starting point of the string.
Y	**int** Specifies the logical *y*-coordinate of the starting point of the string.
lpString	**LPSTR** Points to the character string that is to be drawn.
nCount	**int** Specifies the number of characters in the string.
nTabPositions	**int** Specifies the number of tab-stop positions in the array to which the *lpnTabStopPositions* points.
lpnTabStopPositions	**LPINT** Points to an array of integers containing the tab-stop positions in pixels. The tab stops must be sorted in increasing order; back tabs are not allowed.
nTabOrigin	**int** Specifies the logical *x*-coordinate of the starting position from which tabs are expanded.



Return value

The return value specifies the dimensions of the string. The height is in the high-order word; the width is in the low-order word.

Comments

If the *nTabPositions* parameter is zero the the *lpnTabStopPositions* parameter is NULL, tabs are expanded to eight average character widths.

If *nTabPositions* is 1, the tab stops will be separated by the distance specified by the first value in the array to which *lpnTabStopPositions* points.

If *lpnTabStopPositions* points to more than a single value, then a tab stop is set for each value in the array, up to the number specified by *nTabPositions*.

The *nTabOrigin* parameter allows an application to call the **TabbedTextOut** function several times for a single line. If the application calls **TabbedTextOut** more than once with the *nTabOrigin* set to the same value each time, the function expands all tabs relative to the position specified by *nTabOrigin*.

TextOut

Syntax BOOL TextOut(hDC, X, Y, lpString, nCount)
function TextOut(DC: HDC; X, Y: Integer; Str: PChar; Count: Integer): Bool;

This function writes a character string on the specified display, using the currently selected font. The starting position of the string is given by the *X* and *Y* parameters.

Parameters

hDC	**HDC** Identifies the device context.
X	**int** Specifies the logical *x*-coordinate of the starting point of the string.
Y	**int** Specifies the logical *y*-coordinate of the starting point of the string.
lpString	**LPSTR** Points to the character string that is to be drawn.
nCount	**int** Specifies the number of characters in the string.

Return value The return value specifies whether or not the string is drawn. It is nonzero if the string is drawn. Otherwise, it is zero.

Comments Character origins are defined to be at the upper-left corner of the character cell.

By default, the current position is not used or updated by this function. However, an application can call the **SetTextAlign** function with the *wFlags* parameter set to TA_UPDATECP to permit Windows to use and update the current position each time the application calls **TextOut** for a given device context. When this flag is set, Windows ignores the *X* and *Y* parameters on subsequent **TextOut** calls.

Throw

Syntax void Throw(lpCatchBuf, nThrowBack)
procedure Throw(var CatchBuf: TCatchBuf; ThrowBack: Integer);

This function restores the execution environment to the values saved in the buffer pointed to by the *lpCatchBuf* parameter. The execution environment is the state of all system registers and the instruction counter. Execution continues at the **Catch** function that copied the environment pointed to by *lpCatchBuf*. The *nThrowBack* parameter is passed as the return value to the **Catch** function. It can be a nonzero value.

Parameters

lpCatchBuf — **LPCATCHBUF** Points to an array that contains the execution environment.

nThrowBack — **int** Specifies the value to be returned to the **Catch** function.

Return value None.

Comments The **Throw** function is similar to the C run-time **LongJmp** function (which is incompatible with the Windows environment).

ToAscii 3.0

Syntax

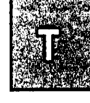
int ToAscii(wVirtKey, wScanCode, lpKeyState, lpChar, wFlags)
function ToAscii(VirtKey, ScanCode: Word; KeyState: PChar; Char: Pointer; Flags: Word): Integer;

This function translates the virtual-key code specified by the *wVirtKey* parameter and the current keyboard state specified by the *lpKeyState* parameter to the corresponding ANSI character or characters.

Parameters

wVirtKey — **WORD** Specifies the virtual-key code to be translated.

wScanCode — **WORD** Specifies the "hardware" raw scan code of the key to be translated. The high-order bit of this value is set if the key is up.



lpKeyState — **LPSTR** Points to an array of 256 bytes, each of which contains the state of one key. If the high-order bit of the byte is set the key is down.

lpChar — **LPVOID** Points to a 32-bit buffer which receives the translated ANSI character or characters.

wFlags — **WORD** The bit 0 flag's menu display.

Return value The return value specifies the number of characters copied to the buffer identified by the *lpChar* parameter. The return value is negative if the key was a dead key. Otherwise, it is one of the following values:

Parameters

2	Two characters were copied to the buffer. This is usually an accent and a dead-key character, when the dead key cannot be translated otherwise.
1	One ANSI character was copied to the buffer.
0	The specified virtual key has no translation for the current state of the keyboard.

Comments The parameters supplied to the **ToAscii** function might not be sufficient to translate the virtual-key code because a previous dead key is stored in the keyboard driver.

Typically, **ToAscii** performs the translation based on the virtual-key code. In some cases, however, the *wScanCode* parameter may be used to distinguish between a key depression or a key release. The scan code is used for translating ALT+NUMBER key combinations.

TrackPopupMenu 3.0

Syntax BOOL TrackPopupMenu(hMenu, wFlags, x, y, nReserved, hWnd, lpReserved)
function TrackPopupMenu(Menu: HMenu; Flags: Word; x, y, Reserved: Integer; Wnd: HWnd; Rect: PRect): Bool;

This function displays a "floating" pop-up menu at the specified location and tracks the selection of items on the pop-up menu. A floating pop-up menu can appear anywhere on the screen. The *hMenu* parameter specifies the handle of the menu to be displayed; the application obtains this handle by calling **CreatePopupMenu** to create a new pop-up menu or by calling **GetSubMenu** to retrieve the handle of a pop-up menu associated with an existing menu item.

Windows sends messages generated by the menu to the window identified by the *hWnd* parameter.

Parameters

hMenu	**HMENU** Identifies the pop-up menu to be displayed.
wFlags	**WORD** Specifies the mouse button that selects items on the menu. If *wFlags* is set to TPM_RIGHTBUTTON, the right mouse button selects items on the menu. Otherwise, the left button selects items on the menu.
x	**int** Specifies the horizontal position in screen coordinates of the left side of the menu on the screen.

y	**int** Specifies the vertical position in screen coordinates of the top of the menu on the screen.
nReserved	**int** Is reserved and must be set to zero.
hWnd	**HWND** Identifies the window which owns the pop-up menu. This window receives all WM_COMMAND messages from the menu.
lpReserved	**LPVOID** Is reserved and must be set to NULL.

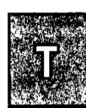
Return value The return value specifies the outcome of the function. It is TRUE if the function is successful. Otherwise, it is FALSE.

TranslateAccelerator

Syntax int TranslateAccelerator(hWnd, hAccTable, lpMsg)
function TranslateAccelerator(Wnd: HWnd; AccTable: THandle; var Msg: TMsg): Integer;

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

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



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

Parameters

hWnd	**HWND** Identifies the window whose messages are to be translated.
hAccTable	**HANDLE** Identifies an accelerator table (loaded by using the **LoadAccelerators** function).
lpMsg	**LPMSG** Points to a message retrieved by using the **GetMessage** or **PeekMessage** function. The message must

be an **MSG** data structure and contain message information from the Windows application queue.

Return value The return value specifies the outcome of the function. It is nonzero if translation occurs. Otherwise, it is zero.

Comments When **TranslateAccelerator** returns nonzero (meaning that the message is translated), the application should *not* process the message again by using the **TranslateMessage** function.

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

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

- The window is disabled.
- The menu item is disabled.
- The command is not in the System menu and the window is minimized.
- A mouse capture is in effect (for more information, see the **SetCapture** function, earlier in this chapter).

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

If an accelerator key stroke that corresponds to a menu item occurs when the window that owns the menu is iconic, no WM_COMMAND message is sent. However, if an accelerator key stroke that does not match any of the items on the window's menu or the System menu occurs, a WM_COMMAND message is sent, even if the window is iconic.

TranslateMDISysAccel 3.0

Syntax BOOL TranslateMDISysAccel(hWndClient, lpMsg)
function TranslateMDISysAccel(Wnd: HWnd; var Msg: TMsg): Bool;

This function processes keyboard accelerators for multiple document interface (MDI) child window System-menu commands. The **TranslateMDISysAccel** function translates WM_KEYUP and WM_KEYDOWN messages to WM_SYSCOMMAND messages. The high-order word of the *lParam* parameter of the WM_SYSCOMMAND message contains the value 1 to differentiate the message from messages sent by menus or controls.

Parameters *hWndClient* **HWND** Identifies the parent MDI client window.

lpMsg **LPMSG** Points to a message retrieved by using the **GetMessage** or **PeekMessage** function. The message must be an **MSG** data structure and contain message information from the Windows application queue.

Return value The return value is TRUE if the function translated a message into a system command. Otherwise, it is FALSE.

TranslateMessage

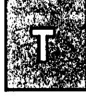
Syntax BOOL TranslateMessage(lpMsg)
function TranslateMessage(var Msg: TMsg): Bool;

This function translates virtual-key messages into character messages, as follows:

- WM_KEYDOWN/WM_KEYUP combinations produce a WM_CHAR or a WM_DEADCHAR message.
- WM_SYSKEYDOWN/WM_SYSKEYUP combinations produce a WM_SYSCHAR or a WM_SYSDEADCHAR message.

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

Parameters *lpMsg* **LPMSG** Points to an **MSG** data structure retrieved through the **GetMessage** or **PeekMessage** function. The structure contains message information from the Windows application queue.



Return value The return value specifies the outcome of the function. It is nonzero if the message is translated (that is, character messages are posted to the application queue). Otherwise, it is zero.

Comments The **TranslateMessage** function does not modify the message given by the *lpMsg* parameter.

TranslateMessage produces WM_CHAR messages only for keys which are mapped to ASCII characters by the keyboard driver.

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

TransmitCommChar

Syntax int TransmitCommChar(nCid, cChar)
function TransmitCommChar(Cid: Integer; Chr: Char): Integer;

This function marks the character specified by the cChar parameter for immediate transmission, by placing it at the head of the transmit queue.

Parameters

nCid	**int** Specifies the communication device to receive the character. The **OpenComm** function returns this value.
cChar	**char** Specifies the character to be transmitted.

Return value The return value specifies the result of the function. It is zero if the function is successful. It is negative if the character cannot be transmitted. A character cannot be transmitted if the character specified by the previous **TransmitCommChar** function has not yet been sent.

UngetCommChar

Syntax int UngetCommChar(nCid, cChar)
function UngetCommChar(Cid: Integer; Chr: Char): Integer;

This function places the character specified by the *cChar* parameter back into the receive queue, making this character the first to be read on a subsequent read from the queue.

Consecutive calls to the **UngetCommChar** function are not allowed. The character placed back into the queue must be read before attempting to place another.

Parameters

nCid	**int** Specifies the communication device to receive the character.
cChar	**char** Specifies the character to be placed in the receive queue.

Return value The return value specifies the outcome of the function. It is zero if the function is successful. It is negative if an error occurs.

UnhookWindowsHook

Syntax BOOL UnhookWindowsHook(nHook, lpfnHook)
function UnhookWindowsHook(Hook: Integer; HookFunc: TFarProc): Bool;

This function removes the Windows hook function pointed to by the *lpfnHook* parameter from a chain of hook functions. A Windows hook function processes events before they are sent to an application's message loop in the WinMain function.

Parameters

nHook — **int** Specifies the type of hook function removed. It may be one of the following values:

Value	Meaning
WH_CALLWNDPROC	Installs a window-function filter.
WH_GETMESSAGE	Installs a message filter.
WH_JOURNALPLAYBACK	Installs a journaling playback filter.
WH_JOURNALRECORD	Installs a journaling record filter.
WH_KEYBOARD	Install a keyboard filter.
WH_MSGFILTER	Installs a message filter.

lpfnHook — **FARPROC** Is the procedure-instance address of the hook function.

Return value The return value specifies the outcome of the function. It is nonzero if the hook function is successfully removed. Otherwise, it is zero.

UnionRect

Syntax int UnionRect(lpDestRect, lpSrc1Rect, lpSrc2Rect)
function UnionRect(var DestRect, Src1Rect, Src2Rect: LPRect): Integer;

This function creates the union of two rectangles. The union is the smallest rectangle that contains both source rectangles.

Parameters

lpDestRect — **LPRECT** Points to the **RECT** data structure that is to receive the new union.

lpSrc1Rect — **LPRECT** Points to a **RECT** data structure that contains a source rectangle.

lpSrc2Rect — **LPRECT** Points to a **RECT** data structure that contains a source rectangle.

Return value The return value specifies the outcome of the function. It is nonzero if the union is not empty. It is zero if the union is empty.

Comments Windows ignores the dimensions of an "empty" rectangle, that is, a rectangle that has no height or has no width.

UnlockData

Syntax HANDLE UnlockData(Dummy)
function UnlockData(Dummy: Integer): THandle;

This macro unlocks the current data segment. It is intended to be used by modules that have moveable data segments.

Parameters *Dummy* **int** Is not used; can be set to zero.

Return value None.

UnlockResource

Syntax BOOL UnlockResource(hResData)
function UnlockResource(ResData: THandle): Bool;

This macro unlocks the resource specified by the *hResData* parameter and decreases the resource's reference count by one.

Parameters *hResData* **HANDLE** Identifies the global memory block to be unlocked.

Return value The return value specifies the outcome of the macro. It is zero if the block's reference count is decreased to zero. Otherwise, it is nonzero.

UnlockSegment

Syntax BOOL UnlockSegment(wSegment)
function UnlockSegment(Segment: Word): THandle;

This function unlocks the segment whose segment address is specified by the *wSegment* parameter. If *wSegment* is –1, the **UnlockSegment** function unlocks the current data segment.

In real mode, or if the segment is discardable, **UnlockSegment** decreases the segment's lock count by one. In protected mode, **UnlockSegment** decreases the lock count of discardable objects and automatic data segments only. The segment is completely unlocked and subject to moving or discarding if the lock count is decreased to zero. Other functions also can affect the lock count of a memory object. See the description of the **GlobalFlags** function for a list of the functions that affect the lock count.

In all cases, each time an application calls **LockSegment** for a segment, it must eventually call **UnlockSegment** for the segment.

Parameters *wSegment* **WORD** Specifies the segment address of the segment to be unlocked. If *wSegment* is –1, the **UnlockSegment** function unlocks the current data segment.

Return value The return value specifies the outcome of the function. It is zero if the segment's lock count was decreased to zero. Otherwise, the return value is nonzero. An application should not rely on the return value to determine the number of times it must subsequently call **UnlockSegment** for the segment.

UnrealizeObject

Syntax BOOL UnrealizeObject(hObject)
function UnrealizeObject(hObject: HBrush): Bool;

If the *hObject* parameter specifies a brush, this function directs GDI to reset the origin of the given brush the next time it is selected.

If *hObject* specifies a logical palette, this function directs GDI to realize the palette as though it had not previously been realized. The next time the application calls the **RealizePalette** function for the specified palette, GDI completely remaps the logical palette to the system palette.

Parameters *hObject* **HANDLE** Identifies the object to be reset.

Return value The return value specifies the outcome of the function. It is nonzero if the function is successful. Otherwise, it is zero.

Comments The **UnrealizeObject** function should not be used with stock objects.

This function must be called whenever a new brush origin is set (by means of the **SetBrushOrigin** function).

A brush specified by the *hObject* parameter must not be the currently selected brush of any display context. A palette specified by *hObject* can be the currently selected palette of a display context.

UnregisterClass 3.0

Syntax BOOL UnregisterClass(lpClassName, hInstance)
function UnregisterClass(ClassName: PChar; Instance: THandle): Bool;

This function removes the window class specified by *lpClassName* from the window-class table, freeing the storage required for the class.

Parameters

lpClassName	**LPSTR** Points to a null-terminated string containing the class name. This class name must have been previously registered by calling the **RegisterClass** function with a valid **hInstance** field in the **WNDCLASS** structure parameter. Predefined classes, such as dialog-box controls, may not be unregistered.
hInstance	**HANDLE** Identifies the instance of the module that created the class.

Return value The return value is TRUE if the function successfully removed the window class from the window-class table. It is FALSE if the class could not be found or if a window exists that was created with the class.

Comments Before using this function, destroy all windows created with the specified class.

UpdateColors 3.0

Syntax int UpdateColors(hDC)
function UpdateColors(DC: HDC): Integer;

This function updates the client area of the device context identified by the *hDC* parameter by matching the current colors in the client area to the system palette on a pixel-by-pixel basis. An inactive window with a realized logical palette may call **UpdateColors** as an alternative to redrawing its client area when the system palette changes. For more information on using color palettes, see *Guide to Programming*.

Parameters

hDC	**HDC** Identifies the device context.

Return value The return value is not used.

Comments **UpdateColors** typically updates a client area faster than redrawing the area. However, because **UpdateColors** performs the color translation based on the color of each pixel before the system palette changed, each call to this function results in the loss of some color accuracy.

UpdateWindow

Syntax void UpdateWindow(hWnd)
procedure UpdateWindow(Wnd: HWnd);

This function updates the client area of the given window by sending a WM_PAINT message to the window if the update region for the window is not empty. The UpdateWindow function sends a WM_PAINT message directly to the window function of the given window, bypassing the application queue. If the update region is empty, no message is sent.

Parameters *hWnd* **HWND** Identifies the window to be updated.

Return value None.

ValidateCodeSegments 3.0

Syntax void ValidateCodeSegments( )
procedure ValidateCodeSegments;

This function outputs debugging information to a terminal if any code segments have been altered by random memory overwrites. It is only available in the debugging version of Windows and is enabled by default. To disable the function, set the **EnableSegmentChecksum** flag in the [kernel] section of WIN.INI to 0. Windows does not validate code segments in protected (standard or 386 enhanced) mode.

Parameters None.

Return value None.

ValidateFreeSpaces

Syntax LPSTR ValidateFreeSpaces( )
function ValidateFreeSpaces: Pointer;

This function (available only in the debugging version of Windows) checks free segments in memory for valid contents. In the debugging version of Windows, the kernel fills all the bytes in free segments with the hexadecimal value CC. This function begins checking for valid contents in the free segment with the lowest address, and continues checking until it finds an invalid byte or until it has determined that all free space contains valid contents. Before calling this function, put the following lines in the WIN.INI file:

```
[kernel]
EnableFreeChecking=1
EnableHeapChecking=1
```

Parameters None.

Return value None.

Comments Windows sends debugging information to the debugging terminal if an invalid byte is encountered and performs a fatal exit.

The [kernel] entries in WIN.INI will cause automatic checking of free memory. Before returning a memory block to the application in response to a **GlobalAlloc** call, Windows checks that memory to make sure it is filled with 0CCH. Before a **GlobalCompact** call, all free memory is checked. Note that using this function slows Windows down system-wide by about 20%.

ValidateRect

Syntax void ValidateRect(hWnd, lpRect)
procedure ValidateRect(Wnd: HWnd; Rect: PRect);

This function validates the client area within the given rectangle by removing the rectangle from the update region of the given window. If the *lpRect* parameter is NULL, the entire window is validated.

Parameters

hWnd	**HWND** Identifies the window whose update region is to be modified.
lpRect	**LPRECT** Points to a **RECT** data structure that contains the rectangle (in client coordinates) to be removed from the update region.

Return value None.

Comments The **BeginPaint** function automatically validates the entire client area. Neither the **ValidateRect** nor **ValidateRgn** function should be called if a portion of the update region needs to be validated before the next WM_PAINT message is generated.

Windows continues to generate WM_PAINT messages until the current update region is validated.

ValidateRgn

Syntax void ValidateRgn(hWnd, hRgn)
procedure ValidateRgn(Wnd: HWnd; Rgn: HRgn);

This function validates the client area within the given region by removing the region from the current update region of the given window. If the *hRgn* parameter is NULL, the entire window is validated.

Parameters

hWnd — **HWND** Identifies the window whose update region is to be modified.

hRgn — **HRGN** Identifies a region that defines the area to be removed from the update region.

Return value None.

Comments The given region must have been created previously by means of a region function (for more information, see Chapter 1, "Window manager interface functions"). The region coordinates are client coordinates.

VkKeyScan

Syntax

int VkKeyScan (cChar)
function VkKeyScan(Chr: Word): Word;

This function translates an ANSI character to the corresponding virtual-key code and shift state for the current keyboard. Applications which send character by means of WM_KEYUP and WM_KEYDOWN messages use this function.

Parameters

cChar — **char** Specifies the character for which the corresponding virtual-key code is to be found.

Return value The VK_ code is returned in the low-order byte and the shift state in the high-order byte. The shift states are:

Value	Meaning
0	No shift.
1	Character is shifted.
2	Character is control character.
6	Charcter is CONTROL+ALT.
7	Character is SHIFT+CONTROL+ALT.
3, 4, 5	A shift key combination that is not used for characters.

If no key is found that translates to the passed ANSI code, a –1 is returned in both the low-order and high-order bytes.

Comments Translations for the numeric keypad (VK_NUMPAD0 through VK_DIVIDE) are ignored. This function is intended to force a translation for the main keyboard only.

WaitMessage

Syntax void WaitMessage( )
procedure WaitMessage;

This function yields control to other applications when an application has no other tasks to perform. The **WaitMessage** function suspends the application and does not return until a new message is placed in the application's queue.

Parameters None.

Return value None.

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

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

WaitSoundState

Syntax int WaitSoundState(nState)
function WaitSoundState(State: Integer): Integer;

This function waits until the play driver enters the specified state.

Parameters *nState* **int** Specifies the state of the voice queues. It can be any one of the following values:

Value	**Meaning**
S_ALLTHRESHOLD	All voices have reached threshold.
S_QUEUEEMPTY	All voice queues are empty and sound drivers turned off.
S_THRESHOLD	A voice queue has reached threshold, and returns voice.

Return value The return value specifies the result of the function. It is zero if the function is successful. If the state is not valid, the return value is S_SERDST.

WindowFromPoint

Syntax HWND WindowFromPoint(Point)
function WindowFromPoint(Point: TPoint): HWnd;

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

Parameters *Point* **POINT** Specifies a **POINT** data structure that defines the point to be checked.

Return value The return value identifies the window in which the point lies. It is NULL if no window exists at the given point.

WinExec 3.0

Syntax WORD WinExec(lpCmdLine, nCmdShow)
function WinExec(CmdLine: PChar; CmdShow: Word): Word;

This function executes the Windows or non-Windows application identified by the *lpCmdLine* parameter. The *nCmdShow* parameter specifies the initial state of the application's main window when it is created.

Parameters *lpCmdLine* **LPSTR** Points to a null-terminated character string that contains the command line (filename plus optional parameters) for the application to be executed. If the *lpCmdLine* string does not contain a directory path, Windows will search for the executable file in this order:

1. The current directory.
2. The Windows directory (the directory containing WIN.COM); the **GetWindowsDirectory** function obtains the pathname of this directory.
3. The Windows system directory (the directory containing such system files as KERNEL.EXE); the **GetSystemDirectory** function obtains the pathname of this directory.
4. The directories listed in the PATH environment variable.

5. The list of directories mapped in a network.

If the application filename does not contain an extension, then .EXE is assumed.

nCmdShow **int** Specifies how a Windows application window is to be shown. See the description of the **ShowWindow** function for a list of the acceptable values for the *nCmdShow* parameter. For a non-Windows application, the PIF file, if any, for the application determines the window state.

Return value The return value specifies whether the function was successful. If the function was successful, the return value is greater than 32. Otherwise, it is a value less than 32 that specifies the error. The following list describes the error values returned by this function:

Value	Meaning
0	Out of memory.
2	File not found.
3	Path not found.
5	Attempt to dynamically link to a task.
6	Library requires separate data segments for each task.
10	Incorrect Windows version.
11	Invalid .EXE file (non-Windows .EXE or error in .EXE image).
12	OS/2 application.
13	DOS 4.0 application.
14	Unknown .EXE type.
15	Attempt in protected (standard or 386 enhanced) mode to load an .EXE created for an earlier version of Windows.
16	Attempt to load a second instance of an .EXE containing multiple, writeable data segments.
17	Attempt in large-frame EMS mode to load a second instance of an application that links to certain nonshareable DLLs already in use.
18	Attempt in real mode to load an application marked for protected mode only.

Comments The **LoadModule** function provides an alternative method for executing a program.

WinHelp 3.0

Syntax BOOL WinHelp(hWnd, lpHelpFile, wCommand, dwData)
function WinHelp(Wnd: HWnd; HelpFile: PChar; Command: Word; Data: Longint): Bool;

This function invokes the Windows Help application and passes optional data indicating the nature of the help requested by the application. The

application specifies the name and, where required, the directory path of the help file which the Help application is to display. See *Tools* for information on creating and using help files.

Parameters

hWnd — **HWND** Identifies the window requesting help.

lpHelpFile — **LPSTR** Points to a null-terminated string containing the directory path, if needed, and the name of the help file which the Help application is to display.

wCommand — **WORD** Specifies the type of help requested. It may be any one of the following values:

Value	Meaning
HELP_CONTEXT	Displays help for a particular context identified by a 32-bit unsigned integer value in *dwData*.
HELP_HELPONHELP	Displays help for using the help application itself. If the *wCommand* parameter is set to HELP_HELPONHELP, **WinHelp** ignores the *lpHelpFile* and *dwData* parameters.
HELP_INDEX	Displays the index of the specified help file. An application should use this value only for help files with a single index. It should not use this value with HELP_SETINDEX.
HELP_KEY	Displays help for a particular key word identified by a string pointed to by *dwData*.
HELP_MULTIKEY	Displays help for a key word in an alternate keyword table.
HELP_QUIT	Notifies the help application that the specified help file is no longer in use.
HELP_SETINDEX	Sets the context specified by the *dwData* parameter as the current index for the help file specified by the *lpHelpFile* parameter. This index remains current until the user accesses a different help file. To help ensure that the correct

index remains set, the application should call **WinHelp** with *wCommand* set to HELP_SETINDEX (with *dwData* specifying the corresponding context identifier) following each call to **WinHelp** with *wCommand* set to HELP_CONTEXT. An application should use this value only for help files with more than one index. It should not use this value with HELP_INDEX.

dwData **DWORD** Specifies the context or key word of the help requested. If *wCommand* is HELP_CONTEXT, *dwData* is a 32-bit unsigned integer containing a context-identifier number. If *wCommand* is HELP_KEY, *dwData* is a long pointer to a null-terminated string that contains a key word identifying the help topic. If *wCommand* is HELP_MULTIKEY, *dwData* is a long pointer to a **MULTIKEYHELP** data structure. Otherwise, *dwData* is ignored and should be set to NULL.

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

Comments The application must call **WinHelp** with *wCommand* set to HELP_QUIT before closing the window that requested the help. The Help application will not actually terminate until all applications that have requested help have called **WinHelp** with *wCommand* set to HELP_QUIT.

WriteComm

Syntax int WriteComm(nCid, lpBuf, nSize)
function WriteComm(Cid: Integer; Buf: PChar; Size: Integer): Integer;

This function writes the number of characters specified by the *nSize* parameter to the communication device specified by the *nCid* parameter from the buffer pointed to by the *lpBuf* parameter.

Parameters

nCid **int** Specifies the device to receive the characters. The **OpenComm** function returns this value.

lpBuf **LPSTR** Points to the buffer that contains the characters to be written.

nSize **int** Specifies the number of characters to write.

Return value The return value specifies the number of characters actually written. When an error occurs, the return value is set to a value less than zero, making the absolute value of the return value the actual number of characters written. The cause of the error can be determined by using the **GetCommError** function to retrieve the error code and status.

Comments The **WriteComm** function will delete data in the transmit queue if there is not enough room in the queue for the additional characters. Applications should check the available space in the transmit queue with the **GetCommError** function before calling **WriteComm**. Also, applications should use the **OpenComm** function to set the size of the transmit queue to an amount no smaller than the size of the largest expected output string.

WritePrivateProfileString 3.0

Syntax BOOL WritePrivateProfileString(lpApplicationName, lpKeyName, lpString, lpFileName)
function WritePrivateProfileString(ApplicationName, KeyName, Str, FileName: PChar): Bool;

This function copies the character string pointed to by the *lpString* parameter into the specified initialization file. It searches the file for the key named by the *lpKeyName* parameter under the application heading specified by the *lpApplicationName* parameter. If there is no match, it adds to the user profile a new string entry containing the key name and the key value specified by the *lpString* parameter. If there is a matching key, the function replaces that key's value with *lpString*.

Parameters

lpApplicationName **LPSTR** Points to an application heading in the initialization file.

lpKeyName **LPSTR** Points to a key name that appears under the application heading in the initialization file.

lpString **LPSTR** Points to the string that contains the new key value.

lpFileName **LPSTR** Points to a null-terminated character string that names the initialization file. If *lpFileName* does not contain a fully qualified pathname for the file, this function searches the Windows directory for the file. If the file does not exist and *lpFileName* does not

W

contain a fully qualified pathname, this function creates the file in the Windows directory. The **WritePrivateProfileString** does not create a file if *lpFileName* contains the fully qualified pathname of a file that does not exist.

Return value The return value specifies the result of the function. It is nonzero if the function is successful. Otherwise, it is zero.

Comments An application should use a private (application-specific) initialization file to record information which affects only that application. This improves both the performance of the application and Windows itself by reducing the amount of information that Windows must read when it accesses the initialization file.

If there is no application field for *lpApplicationName*, this function creates a new application field and places an appropriate key-value line in that field of the initialization file.

A string entry in the initialization file has the following form:

```
[application name]
keyname = string
:
```

An application can also call **WritePrivateProfileString** to delete lines from its private initialization file. If *lpString* is NULL, the function deletes the entire line identified by the *lpKeyName* parameter. If *lpString* points to a null string, the function deletes only the value; the key name remains in the file. If *lpKeyName* is NULL, the function deletes the entire section identified by the *lpApplicationName* parameter; however, the function does not delete any lines beginning with the semicolon (;) comment character.

WriteProfileString

Syntax BOOL WriteProfileString(lpApplicationName, lpKeyName, lpString)
function WriteProfileString(ApplicationName, KeyName, Str: PChar): Bool;

This function copies the character string pointed to by the *lpString* parameter into the Windows initialization file, WIN.INI. It searches WIN.INI for the key named by the *lpKeyName* parameter under the application heading specified by the *lpApplicationName* parameter. If there is no match, it adds to the user profile a new string entry containing the key name and the key value specified by the *lpString* parameter. If there is a matching key, the function replaces that key's value with *lpString*.

Parameters

lpApplicationName	**LPSTR** Points to an application heading in WIN.INI.
lpKeyName	**LPSTR** Points to a key name that appears under the application heading WIN.INI.
lpString	**LPSTR** Points to the string that contains the new key value.

Return value The return value specifies the result of the function. It is nonzero if the function is successful. Otherwise, it is zero.

Comments If there is no match for *lpApplicationName*, this function creates a new application field and adds the string pointed to by *lpString*.

A string entry in WIN.INI has the following form:

```
[application name]
keyname = string
:
```

An application can also call **WriteProfileString** to delete lines from WIN.INI. If *lpString* is NULL, the function deletes the entire line identified by the *lpKeyName* parameter. If *lpString* points to a null string, the function deletes only the value; the key name remains in the file. If *lpKeyName* is NULL, the function deletes the entire section identified by the *lpApplicationName* parameter; however, the function does not delete any lines beginning with the semicolon (;) comment character.

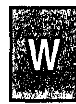
wsprintf 3.0

Syntax int wsprintf(lpOutput, lpFormat[[, argument]]. . .)

This function formats and stores a series of characters and values in a buffer. Each argument (if any) is converted and output according to the corresponding format specification in the format string. The function appends a NULL to the end of the characters written, but the return value does not include the terminating null character in its character count.

Parameters

lpOutput	**LPSTR** Points to a null-terminated character string to receive the formatted output.
lpFormat	**LPSTR** Points to a null-terminated character string that contains the format-control string. In addition to ordinary ASCII characters, a format specification for each argument appears in this string. See the following "Comments" section for more information on the format specification.



argument Is one or more optional arguments. The number and type of *argument* parameters depends on the corresponding format-control character sequences in *lpFormat*.

Return value The return value is the number of characters stored in *lpOutput*, not counting the terminating NULL. If an error occurs, the function returns a value less than the length of *lpFormat*.

Comments The format-control string contains format specifications that determine the output format for the arguments which follow the *lpFormat* parameter. Format specifications, discussed below, always begin with a percent sign (%). If a percent sign is followed by a character that has no meaning, such as a format field, the character is output as is. For example, %% produces a single percent-sign character.

The format-control string is read from left to right. When the first format specification (if any) is encountered, it causes the value of the first argument after the format-control string to be converted and output according to the format specification. The second format specification causes the second argument to be converted and output, and so on. If there are more arguments than there are format specifications, the extra arguments are ignored. The results are undefined if there are not enough arguments for all of the format specifications.

A format specification has the following form:

%[[–]][[#]][[0]][[*width*]][[.*precision*]]*type*

Each field of the format specification is a single character or a number signifying a particular format option. The *type* characters, which appear after the last optional format field, determine whether the associated argument is interpreted as a character, a string, or a number. The simplest format specification contains only the percent sign and a type character (for example, %s). The optional fields control other aspects of the formatting. The following shows the optional and required fields and their meaning:

Parameters

– Pad the output with blanks or zeroes to the right to fill the field width, justifying the output to the left. If this field is omitted, the output is padded to the left, justifying the output to the right.

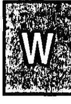
\# Prefix hexadecimal values with 0x (lowercase) or OX (uppercase).

0 Pad the output value with zeroes to fill the field width. If this field is omitted, the output value is padded with blank spaces.

width	Output the specified minimum number of characters. The *width* field is a nonnegative integer. The width specification never causes a value to be truncated; if the number of characters in the output value is greater than the specified width, or if the *width* field is not present, all characters of the value are printed, subject to the precision specification.
precision	Output the specified minimum number of digits. If the number of digits in the argument is less than the specified precision, the output value is padded on the left with zeroes. The value is not truncated when the number of digits exceeds the specified precision. If the specified precision is 0, omitted entirely, or if the period (.) appears without a number following it, the precision is set to 1. For strings, output the specified maximum number of characters.
type	Output the corresponding argument as a character, string, or a number. This field may be any of the following character sequences:

Sequence	**Meaning**
s	Insert a string argument referenced by a long pointer. The argument corresponding to this sequence *must* be passed as a long pointer (**LPSTR**).
c	Insert a single character argument. The **wsprintf** function ignores character arguments with a numerical value of zero.
d, i	Insert a signed decimal integer argument.
ld, li	Insert a long signed decimal integer argument.
u	Insert an unsigned integer argument.
lu	Insert a long unsigned integer argument.
x, X	Insert an unsigned hexadecimal integer argument in lowercase or uppercase.
lx, lX	Insert a long unsigned hexadecimal integer argument in lowercase or uppercase.

Unlike all other Windows functions, **wsprintf** uses the C calling convention (**cdecl**), rather than the Pascal calling convention. As a result, it is the caller's responsibility to pop arguments off the stack, and arguments are pushed in reverse order (that is, the *lpOutput* parameter is pushed last, to the lowest address). In C-language modules, the C compiler performs this task.



wvsprintf 3.0

Syntax

int wvsprintf(lpOutput, lpFormat, lpArglist)
function wvsprintf(DestStr, Format, ArgList: PChar): Integer;

This function formats and stores a series of characters and values in a buffer. The items pointed to by the argument list are converted and output according to the corresponding format specification in the format string.

The function appends a NULL to the end of the characters written, but the return value does not include the terminating null character in its character count.

Parameters

lpOutput **LPSTR** Points to a null-terminated character string to receive the formatted output.

lpFormat **LPSTR** Points to a null-terminated character string that contains the format-control string. In addition to ordinary ASCII characters, a format specification for each argument appears in this string. See the description of the **wsprintf** function, earlier in this chapter, for more information on the format specification.

lpArglist **LPSTR** Points to an array of words, each of which specifies an arguement for the format-control string. The number, type and interpretation of the arguments depend on the corresponding format-control character sequences in *lpFormat*. Each character or word-sized integer (%c, %d, %x, %i) requires one word in *lpArglist*. Long integers (%ld, %li, %lx) require two words, the low-order word of the integer followed by the high-order word. A string (%s) requires two words, the offset followed by the segment (which together make up a far pointer).

Return value

The return value is the number of characters stored in *lpOutput*, not counting the terminating NULL. If an error occurs, the function returns a value less than the length of *lpFormat*.

Yield

Syntax

```
void Yield( )
function Yield: Bool;
```

This function halts the current task and starts any waiting task.

Parameters None.

Return value None.

Comments Applications that contain windows should use a **DispatchMessage**, **PeekMessage**, or **TranslateMessage** loop rather than calling the **Yield** function directly. The **PeekMessage** loop handles message synchronization properly and yields at the appropriate times.

PART 2

Windows messages

Part 2 provides reference information on Windows messages. Windows messages allow Windows applications to communicate with each other and with the Windows system within a nonpreemptive multitasking environment.

C H A P T E R

5

Messages overview

See Chapter 1, "Window manager interface functions," for an explanation of sending and receiving messages.

This chapter describes groups of related Microsoft Windows messages. Each section states the purpose of the message group, lists the messages in the group, and describes each message.

This chapter lists the following categories of Windows messages:

- Window-management messages
- Initialization messages
- Input messages
- System messages
- Clipboard messages
- System-information messages
- Control messages
- Notification messages
- Scroll-bar messages
- Nonclient-area messages
- Multiple document interface messages

Window-management messages

Window-management messages are sent by Windows to an application when the state of a window changes. The following list briefly describes each window-management message:

Message	Description
WM_ACTIVATE	Sent when a window becomes active or inactive.
WM_ACTIVATEAPP	Sent when the window being activated belongs to a different application than the window that was previously active.
WM_CANCELMODE	Cancels any mode the system is in, such as one that tracks the mouse in a scroll bar or moves a window. Windows sends the WM_CANCELMODE message when an application displays a message box.
WM_CHILDACTIVATE	Notifies a child window's parent window when the **SetWindowPos** function moves a child window.
WM_CLOSE	Sent whenever the window is closed.
WM_CREATE	Sent when the **CreateWindow** function is called.
WM_CTLCOLOR	Sent to the parent window of a predefined control or message box when the control or message box is about to be drawn.
WM_DESTROY	Sent when the **DestroyWindow** function is called, after the window has been removed from the screen.
WM_ENABLE	Sent after a window has been enabled or disabled.
WM_ENDSESSION	Tells an application that has responded nonzero to a WM_QUERYENDSESSION message whether the session is actually being ended.
WM_ENTERIDLE	Informs a window that a dialog box or menu is displayed and waiting for user action.
WM_ERASEBKGND	Sent when the window background needs to be erased.
WM_GETDLGCODE	Sent to an input procedure associated with a control.
WM_GETMINMAXINFO	Retrieves the maximized size of the window, the minimum or maximum tracking size of the window, and the maximized position of the window.
WM_GETTEXT	Copies the text that corresponds to a window.
WM_GETTEXTLENGTH	Retrieves the length (in bytes) of the text associated with a window.
WM_ICONERASEBKGND	Sent to an iconic window with a class icon when the background of the icon needs to be erased.
WM_KILLFOCUS	Sent immediately before a window loses the input focus.
WM_MENUCHAR	Notifies the window that owns the menu when the user presses a menu mnemonic character that doesn't match any of the predefined mnemonics in the current menu.
WM_MENUSELECT	Notifies a window that the user has selected a menu item.
WM_MOVE	Sent when a window is moved.

WM_PAINT	Sent whenever Windows or an application makes a request to repaint a portion of an application's window.
WM_PAINTICON	Sent whenever Windows or an application makes a request to repaint a portion of an application's minimized (iconic) window. WM_PARENTNOTIFY
WM_PARENTNOTIFY	Sent to the parent of a child window when the child window is created or destroyed.
WM_QUERYDRAGICON	Sent when the user is about to drag a minimized (iconic) window.
WM_QUERYENDSESSION	Sent when the user chooses the End Session command.
WM_QUERYNEWPALETTE	Sent when a window is about to receive the input focus to allow it to realize its logical color palette.
WM_QUERYOPEN	Sent to an icon when the user requests that the icon be opened into a window.
WM_QUIT	Indicates a request to terminate an application.
WM_SETFOCUS	Sent after a window receives the input focus.
WM_SETFONT	Changes the font used by a control for drawing text.
WM_SETREDRAW	Sets or clears the redraw flag, which determines whether or not updates to a control are displayed.
WM_SETTEXT	Sets the text of a window.
WM_SHOWWINDOW	Sent whenever a window is to be hidden or shown.
WM_SIZE	Sent after the size of a window has been changed.

Initialization messages

Initialization messages are sent by Windows when an application creates a menu or dialog box. The following list briefly describes each initialization message:

Message	**Description**
WM_INITDIALOG	Sent immediately before a dialog box is displayed.
WM_INITMENU	Requests that a menu be initialized.
WM_INITMENUPOPUP	Sent immediately before a pop-up menu is displayed.

Input messages

Input messages are sent by Windows when an application receives input through the mouse, keyboard, scroll bars, or system timer. The following list briefly describes each input message:

Message	Description
WM_CHAR	Results when a WM_KEYUP and a WM_KEYDOWN message are translated.
WM_CHARTOITEM	Sent by a list box with the LBS_WANTKEYBOARDINPUT style to its owner in response to a WM_CHAR message.
WM_COMMAND	Sent when the user selects an item from a menu, when a control passes a message to its parent window, or when an accelerator key stroke is translated.
WM_DEADCHAR	Results when a WM_KEYUP and a WM_KEYDOWN message are translated.
WM_HSCROLL	Sent when the user clicks the horizontal scroll bar with the mouse.
WM_KEYDOWN	Sent when a nonsystem key is pressed.
WM_KEYUP	Sent when a nonsystem key is released.
WM_LBUTTONDBLCLK	Sent when the user double-clicks the left mouse button.
WM_LBUTTONDOWN	Sent when the user presses the left mouse button.
WM_LBUTTONUP	Sent when the user releases the left mouse button.
WM_MBUTTONDBLCLK	Sent when the user double-clicks the middle mouse button.
WM_MBUTTONDOWN	Sent when the user presses the middle mouse button.
WM_MBUTTONUP	Sent when the user releases the middle mouse button.
WM_MOUSEACTIVATE	Sent when the cursor is in an inactive window and any mouse button is pressed.
WM_MOUSEMOVE	Sent when the user moves the mouse.
WM_RBUTTONDBLCLK	Sent when the user double-clicks the right mouse button.
WM_RBUTTONDOWN	Sent when the user presses the right mouse button.
WM_RBUTTONUP	Sent when the user releases the right mouse button.
WM_SETCURSOR	Sent when mouse input is not captured and the mouse causes cursor movement within a window.
WM_TIMER	Sent when the time limit set for a given timer has elapsed.
WM_VKEYTOITEM	Sent by a list box with the LBS_WANTKEYBOARDINPUT style to its owner in response to a WM_CHAR message.
WM_VSCROLL	Sent when the user clicks the vertical scroll bar with the mouse.

System messages

System messages are sent by Windows to an application when a user accesses a window's System menu, scroll bars, or size box. Although an

application can process these messages, most applications pass them on to the **DefWindowProc** function for default processing. The following list briefly describes each system message:

Message	Description
WM_SYSCHAR	Results when a WM_SYSKEYUP and a WM_SYSKEYDOWN message are translated.
WM_SYSCOMMAND	Sent when the user selects a command from the System menu.
WM_SYSDEADCHAR	Results when a WM_SYSKEYUP and a WM_SYSKEYDOWN message are translated.
WM_SYSKEYDOWN	Sent when the user holds down the ALT key and then presses another key.
WM_SYSKEYUP	Sent when the user releases a key that was pressed while the ALT key was held down.

Clipboard messages

Clipboard messages are sent by Windows to an application when other applications try to access a window's clipboard. The following list briefly describes each clipboard message:

Message	Description
WM_ASKCBFORMATNAME	Requests the name of the CF_OWNERDISPLAY format.
WM_CHANGECBCHAIN	Notifies viewing-chain members of a change in the chain.
WM_DESTROYCLIPBOARD	Signals that the contents of the clipboard are being destroyed.
WM_DRAWCLIPBOARD	Signals an application to notify the next application in the chain of a clipboard change.
WM_HSCROLLCLIPBOARD	Requests horizontal scrolling for the CF_OWNERDISPLAY format.
WM_PAINTCLIPBOARD	Requests painting of the CF_OWNERDISPLAY format.
WM_RENDERALLFORMATS	Notifies the clipboard owner that it must render the clipboard data in all possible formats.
WM_RENDERFORMAT	Notifies the clipboard owner that it must format the last data copied to the clipboard.
WM_SIZECLIPBOARD	Notifies the clipboard owner that the clipboard application's window size has changed.
WM_VSCROLLCLIPBOARD	Requests vertical scrolling for the CF_OWNERDISPLAY format.

System information messages

System-information messages are sent by Windows when an application or a user makes a system-wide change that affects other applications. The following list briefly describes each system-information message:

Message	Description
WM_COMPACTING	Sent to all top-level windows when Windows requires too much system time compacting memory, indicating that system memory is low.
WM_DEVMODECHANGE	Sent to all top-level windows when the user changes device-mode settings.
WM_FONTCHANGE	Sent when the pool of font resources changes.
WM_PALETTECHANGED	Notifies all windows that the system color palette has changed.
WM_SPOOLERSTATUS	Sent from Print Manager whenever a job is added to or removed from the Print Manager queue.
WM_SYSCOLORCHANGE	Sent to all top-level windows when a change is made in the system color setting.
WM_TIMECHANGE	Sent when an application makes a change or set of changes to the system time.
WM_WININICHANGE	Sent when the Windows initialization file, WIN.INI, changes.

Control messages

Control messages are predefined window messages that direct a control to carry out a specified task. Applications send control messages to a control by using the **SendMessage** function. The control carries out the specified task and returns a value that indicates the result.

The following messages apply to all controls:

Message	Description
WM_NEXTDLGCTL	Sent to a dialog box's window function, to alter the control focus.
WM_GETFONT	Retrieves the current font used by a control for drawing text.
WM_SETFONT	Changes the font used by a control for drawing text.

The sections "Button-control messages" through "Owner draw-control messages" briefly describe the control messages for each control class.

Button-control messages

Button-control messages are sent by an application to a button control. The following list briefly describes each button-control message:

Message	Description
BM_GETCHECK	Determines whether a radio button or check box is checked.
BM_GETSTATE	Returns nonzero if the cursor is over the button and the user presses the mouse button or the SPACEBAR.
BM_SETCHECK	Checks or removes the checkmark from a radio button or check box.
BM_SETSTATE	Highlights a button or check box.
BM_SETSTYLE	Alters the style of a button.
DM_GETDEFID	Retrieves the ID of the default pushbutton control for a dialog box.
DM_SETDEFID	Changes the default push-button control ID for a dialog box.

Edit-control messages

Edit-control messages are sent by an application to an edit control. In addition to the messages described below, the WM_ENABLE, WM_GETTEXT, WM_GETTEXTLENGTH, WM_KILLFOCUS, WM_SETFOCUS, WM_SETREDRAW, and WM_SETTEXT window messages can be used. The following list briefly describes each edit-control message:

Message	Description
EM_CANUNDO	Determines whether or not an edit control can respond correctly to an EM_UNDO message.
EM_EMPTYUNDOBUFFER	Disables an edit control's ability to undo the last edit.
EM_FMTLINES	Directs the edit control to add or remove the end-of-line character from wordwrapped text lines.
EM_GETHANDLE	Returns the data handle of the buffer used to hold the contents of the control window.
EM_GETLINE	Copies a line from the edit control.
EM_GETLINECOUNT	Returns the number of lines of text in the edit control.
EM_GETMODIFY	Returns the current value of the modify flag for a given edit control. The flag is set by the control if the user enters or modifies text within the control.
EM_GETRECT	Returns the formatting rectangle of the edit control.
EM_GETSEL	Returns the starting and ending character positions of the current selection.

EM_LIMITTEXT	Limits the length of the text (in bytes) the user may enter.
EM_LINEFROMCHAR	Returns the line number of the line that contains the character whose position (indexed from the beginning of the text) is specified by the *wParam* parameter.
EM_LINEINDEX	Returns the number of character positions that occur before the first character in a given line.
EM_LINELENGTH	Returns the length of a line (in bytes) in the edit control's text buffer.
EM_LINESCROLL	Scrolls the contents of the edit control by the given number of lines.
EM_REPLACESEL	Replaces the current selection with new text.
EM_SETHANDLE	Establishes the text buffer used to hold the contents of the edit-control window.
EM_SETMODIFY	Sets the modify flag for a given edit control.
EM_SETPASSWORDCHAR	Changes the password character for an edit control created with the ES_PASSWORD styles.
EM_SETRECT	Sets the formatting rectangle for an edit control.
EM_SETRECTNP	Identical to EM_SETRECT, except that the control is *not* repainted.
EM_SETSEL	Selects all characters in the current text that are within the starting and ending character positions given by the *lParam* parameter.
EM_SETTABSTOPS	Sets tab-stop positions in a multiline edit control.
EM_SETWORDBREAK	Informs a multiline edit control that Windows has replaced the default word-break function with an application-supplied word-break function.
EM_UNDO	Undoes the last edit in an edit control.
WM_CLEAR	Deletes the current selection.
WM_COPY	Sends the current selection to the clipboard in CF_TEXT format.
WM_CUT	Sends the current selection to the clipboard in CF_TEXT format, and then deletes the selection from the control window.
WM_PASTE	Inserts the data from the clipboard into the control window at the current cursor position.
WM_UNDO	Undoes the previous action.

List-box messages

List-box messages are sent by an application to a list box. The following list briefly describes each list-box message:

Message	Description
LB_ADDSTRING	Adds a string to the list box.
LB_DELETESTRING	Deletes a string from the list box.
LB_DIR	Adds a list of the files from the current directory to the list box.

LB_FINDSTRING	Finds the first string in the list box which matches prefix text.
LB_GETCOUNT	Returns a count of the number of items in the list box.
LB_GETCURSEL	Returns the index of the currently selected item, if any.
LB_GETHORIZONTALEXTENT	Retrieves the width by which a list box can be scrolled horizontally.
LB_GETITEMDATA	Retrieves a 32-bit value associated with an item in an owner-draw list box.
LB_GETITEMRECT	Retrieves the coordinates of the rectangle that bounds a list-box item.
LB_GETSEL	Returns the selection state of an item.
LB_GETSELCOUNT	Returns the total number of selected items in a multiselection list box.
LB_GETSELITEMS	Retrieves the indexes of the selected items in a multiselection list box.
LB_GETTEXT	Copies a string from the list box into a buffer.
LB_GETTEXTLEN	Returns the length of a string in the list box.
LB_GETTOPINDEX	Returns the index of the first visible item in a list box.
LB_INSERTSTRING	Inserts a string in the list box.
LB_RESETCONTENT	Removes all strings from a list box and frees any memory allocated for those strings.
LB_SELECTSTRING	Changes the current selection to the first string that has the specified prefix.
LB_SELITEMRANGE	Selects one or more consecutive items in a multiple-selection list box.
LB_SETCOLUMNWIDTH	Sets the width in pixels of all columns in a multicolumn list box.
LB_SETCURSEL	Selects a string and scrolls it into view, if necessary.
LB_SETHORIZONTALEXTENT	Sets the width by which a list box can be scrolled horizontally.
LB_SETITEMDATA	Sets a 32-bit value associated with an item in an owner-draw list box.
LB_SETSEL	Sets the selection state of a string.
LB_SETTABSTOPS	Sets tab-stop positions in a list box.
LB_SETTOPINDEX	Sets the first visible item in a list box to the item identified by an index.

Combo-box messages

Combo-box messages are sent by an application to a combo box. The following list briefly describes each combo-box message:

Message	Description
CB_ADDSTRING	Adds a string to the list box of a combo box.
CB_DELETESTRING	Deletes a string from the list box of a combo box.

CB_DIR	Adds a list of the files from the current directory to the combo box.
CB_FINDSTRING	Finds the first string in the combo-box list box which matches a prefix.
CB_GETCOUNT	Returns a count of the number of items in the combo box.
CB_GETCURSEL	Returns the index of the currently selected item, if any.
CB_GETEDITSEL	Returns the starting and ending positions of the selected text in the edit control of a combo box.
CB_GETITEMDATA	Retrieves a 32-bit value associated with an item in an owner-draw combo box.
CB_GETLBTEXT	Copies a string from the list box of a combo box into a buffer.
CB_GETLBTEXTLEN	Returns the length of a string in the list box of a combo box.
CB_INSERTSTRING	Inserts a string in the combo box.
CB_LIMITTEXT	Limits the length of the text that the user may enter into the edit control of a combo box.
CB_RESETCONTENT	Removes all strings from a combo box and frees any memory allocated for those strings.
CB_SELECTSTRING	Changes the current selection to the first string that has the specified prefix. The text in the edit control is changed to reflect the new selection.
CB_SETCURSEL	Selects a string and scrolls it into view, if necessary.
CB_SETEDITSEL	Selects all characters in the edit control that are within specified starting and ending positions.
CB_SETITEMDATA	Sets a 32-bit value associated with an item in an owner-draw combo box.
CB_SHOWDROPDOWN	Shows or hides a drop-down list box in a combo box.

Owner draw-control messages

Owner draw-control messages notify the owner of a control created with the OWNERDRAW style that the control needs to be drawn and to provide information about the drawing required. The following list briefly describes these messages:

Message	Description
WM_COMPAREITEM	Determines which of two items sorts above the other in a sorted owner-draw list box or combo box.
WM_DELETEITEM	Indicates that an item in an owner-draw list box or combo box has been deleted.
WM_DRAWITEM	Indicates that an owner-draw control needs to be redrawn.
WM_MEASUREITEM	Requests the dimensions of an owner-draw combo box, list box, or menu item.

Notification messages

Notification messages notify a control's parent window of actions that occur within a control. The sections "Button notification codes" through "Combo-box notification codes" briefly describe the notification messages for each notification class.

Controls use the WM_COMMAND message to notify the parent window of actions that occur within the control. The *wParam* parameter of the WM_COMMAND message contains the control ID; the low-order word of the *lParam* parameter contains the control-window handle; and the high-order word of *lParam* contains the control notification code.

Button notification codes

The following notification codes apply to buttons:

Message	Description
BN_CLICKED	Indicates that the button has been clicked.
BN_DOUBLECLICKED	Indicates that the user has double-clicked an owner-draw or radio button.

Edit-control notification codes

The following notification codes apply to edit controls:

Message	Description
EN_CHANGE	Indicates that the user has taken some action that may have changed the content of the text.
EN_ERRSPACE	Indicates that the edit control is out of space.
EN_HSCROLL	Indicates that the user has clicked the edit control's horizontal scroll bar with the mouse; the parent window is notified before the screen is updated.
EN_KILLFOCUS	Indicates that the edit control has lost the input focus.
EN_MAXTEXT	Specifies that the current insertion has exceeded a specified number of characters for the edit control.
EN_SETFOCUS	Indicates that the edit control has obtained the input focus.
EN_UPDATE	Specifies that the edit control will display altered text.
EN_VSCROLL	Indicates that the user has clicked the edit control's vertical scroll bar with the mouse; the parent window is notified before the screen is updated.

List-box notification codes

The following notification codes apply only to list-box controls that have LBS_NOTIFY style:

Message	Description
LBN_DBLCLK	Sent when the user double-clicks a string with the mouse.
LBN_ERRSPACE	Sent when the system is out of memory.
LBN_KILLFOCUS	Indicates that a list box has lost input focus.
LBN_SELCHANGE	Sent when the selection has been changed.
LBN_SETFOCUS	Indicates that the list box has received input focus.

Combo-box notification codes

The following notification codes apply to combo boxes:

Message	Description
CBN_DBLCLK	Sent when the user double-clicks a string with the mouse.
CBN_DROPDOWN	Informs the owner of the combo box that its list box is about to be dropped down.
CBN_EDITCHANGE	Indicates that the user has altered text in the edit control.
CBN_EDITUPDATE	Indicates that the edit control will display altered text.
CBN_ERRSPACE	Sent when the system is out of memory.
CBN_KILLFOCUS	Indicates that a combo box has lost input focus.
CBN_SELCHANGE	Sent when the selection has been changed.
CBN_SETFOCUS	Indicates that the combo box has received input focus.

Scroll-bar messages

There are two messages in the scroll-bar group: WM_HSCROLL and WM_VSCROLL. Scroll-bar controls send these messages to their parent windows whenever the user clicks in the control. The *wParam* parameter contains the same values as those defined for the scrolling messages of a standard window. The high-order word of the *lParam* parameter contains the window handle of the scroll-bar control.

Nonclient-area messages

Nonclient-area messages are sent by Windows to create and maintain the nonclient area of an application's window. Normally, applications do not

process these messages, but send them on to the **DefWindowProc** function for processing. The following list briefly describes each nonclient-area message:

Message	Description
WM_NCACTIVATE	Sent to a window when its caption bar or icon needs to be changed to indicate an active or inactive state.
WM_NCCALCSIZE	Sent when the size of a window's client area needs to be calculated.
WM_NCCREATE	Sent prior to the WM_CREATE message when a window is first created.
WM_NCDESTROY	Sent after the WM_DESTROY message.
WM_NCHITTEST	Sent to the window that contains the cursor (unless a window has captured the mouse).
WM_NCLBUTTONDBLCLK	Sent to a window when the left mouse button is double-clicked while the cursor is in a nonclient area of the window.
WM_NCLBUTTONDOWN	Sent to a window when the left mouse button is pressed while the cursor is in a nonclient area of the window.
WM_NCLBUTTONUP	Sent to a window when the left mouse button is released while the cursor is in a nonclient area of the window.
WM_NCMBUTTONDBLCLK	Sent to a window when the middle mouse button is double-clicked while the cursor is in a nonclient area of the window.
WM_NCMBUTTONDOWN	Sent to a window when the middle mouse button is pressed while the cursor is in a nonclient area of the window.
WM_NCMBUTTONUP	Sent to a window when the left mouse button is released while the cursor is in a nonclient area of the window.
WM_NCMOUSEMOVE	Sent to a window when the cursor is moved in a nonclient area of the window.
WM_NCPAINT	Sent to a window when its border needs painting.
WM_NCRBUTTONDBLCLK	Sent to a window when the right mouse button is double-clicked while the cursor is in a nonclient area of the window.
WM_NCRBUTTONDOWN	Sent to a window when the right mouse button is pressed while the cursor is in a nonclient area of the window.
WM_NCRBUTTONUP	Sent to a window when the right mouse button is released while the cursor is in a nonclient area of the window.

Multiple document interface messages

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

Message	Description
WM_MDIACTIVATE	Activates a child window.
WM_MDICASCADE	Arranges child windows in a cascade format.
WM_MDICREATE	Creates a child window.
WM_MDIDESTROY	Closes a child window.
WM_MDIGETACTIVE	Returns the current active MDI child window.
WM_MDIICONARRANGE	Arranges all minimized child windows.
WM_MDIMAXIMIZE	Maximizes an MDI child window.
WM_MDINEXT	Activates the next child window.
WM_MDIRESTORE	Restores a child window from a maximized or minimized state.
WM_MDISETMENU	Replaces the menu of an MDI frame window, the Window pop-up menu, or both.
WM_MDITILE	Arranges all child windows in a tiled format.

Topic	Reference
Message-processing functions	*Reference, Volume 1*: Chapter 1, "Window manager interface functions"
Function descriptions	*Reference, Volume 1*: Chapter 4, "Functions directory"
Message descriptions	*Reference, Volume 1*: Chapter 6, "Messages directory"
Windows data types and structures	*Reference, Volume 2*: Chapter 7, "Data types and structures"
Dynamic data exchange	*Reference, Volume 2*: Chapter 15, "Windows DDE protocol definition" *Guide to Programming*: Chapter 22, "Dynamic data exchange"
General information on Windows programming	*Guide to Programming*: Chapter 1, "An overview of the Windows environment"

C H A P T E R

6

Messages directory

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

Some messages return values that contain information about the success of the message or other data needed by an application. To obtain the return value, the application must call **SendMessage** to send the message to a window. This function does not return until the message has been processed. If the application does not require the return value of the message, it may call **PostMessage** to send the message. This function places a message in a window's application queue and then returns immediately. If a message does not have a return value, then the application may use either function to send the message, unless indicated otherwise in the message description.

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

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

can also be used with **HIWORD** and **LOWORD** to access any of the bytes. Casting can also be used.

There are four ranges of message numbers, as shown in the following list:

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

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

Message numbers in the second range (WM_USER to 7FFF) can be defined and used by an application to send messages within the application. These messages should *not* be sent to other applications unless the applications have been designed to exchange messages and to attach the same meaning to the message numbers.

Message numbers in the third range (8000 to BFFF) are reserved for future use by Windows.

Message numbers in the fourth range (C000 to FFFF) are defined at run time when an application calls the **RegisterWindowMessage** function to obtain a message number for a string. All applications that register the identical string can use the associated message number for exchanging messages with each other. The actual message number, however, is not a constant and cannot be assumed to be the same in different window sessions.

This chapter lists messages in alphabetical order. For more information about messages, see Chapter 5, "Messages overview."

BM_GETCHECK

This message determines whether a radio button or check box is checked.

Parameters *wParam* Is not used.

lParam Is not used.

Return value The return value is nonzero if the radio button or check box is checked. Otherwise, it is zero. The BM_GETCHECK message always returns zero for a push button.

BM_GETSTATE

This message determines the state of a button control when the user presses a mouse button or the SPACEBAR.

Parameters *wParam* Is not used.

lParam Is not used.

Return value The BM_GETSTATE message returns a nonzero value if one of the following occurs:

- A push button is highlighted.
- The user presses a mouse button or the SPACEBAR when a button has the input focus.
- The user presses a mouse button when the cursor is over a button.

Otherwise, BM_GETSTATE returns zero.

BM_SETCHECK

This message checks or removes the checkmark from a radio button or check box.

Parameters *wParam* Specifies whether to place or remove a checkmark inside the button or box. If the *wParam* parameter is nonzero, a checkmark is placed; if it is zero, the checkmark (if any) is removed. For three-state buttons, if *wParam* is 1, a checkmark is placed beside the button. If *wParam* is 2, the button is grayed. If *wParam* is zero, the button is returned to its normal state (no checkmark or graying).

lParam	Is not used.

Comments The BM_SETCHECK message has no effect on push buttons.

BM_SETSTATE

This message displays a button or check box.

Parameters

wParam	Specifies the highlighting action to be taken. If the *wParam* parameter is nonzero, the button is highlighted (the interior is drawn using inverse video). If *wParam* is zero, the button is drawn in its regular state.
lParam	Is not used.

Comments Push buttons cannot be highlighted.

BM_SETSTYLE

This message alters the style of buttons. If the style contained in the *wParam* parameter differs from the existing style, the button is redrawn in the new style.

Parameters

wParam	Specifies the style value. For a complete description of possible button styles, see Table 6.1, "Button styles."
lParam	Specifies whether or not the buttons are to be redrawn. If *lParam* is zero, the buttons will not be redrawn. If *lParam* is nonzero, they will be redrawn.

Comments Table 6.1 describes the available button styles:

Table 6.1 Button styles

Value	Meaning
BS_AUTOCHECKBOX	Identical to BS_CHECKBOX, except that the button automatically toggles its state whenever the user clicks it.
BS_AUTORADIOBUTTON	Identical to BS_RADIOBUTTON, except that the button is checked, the application is notified by BN_CLICKED, and the checkmarks are removed from all other radio buttons in the group.
BS_AUTO3STATE	Identical to BS_3STATE, except that the button automatically toggles its state when the user clicks it.
BS_CHECKBOX	Designates a box that may be checked; its border is bold when the user clicks the button. Any text appears to the right of the box.

Table 6.1: Button styles (continued)

BS_DEFPUSHBUTTON	Designates a button with a bold border. This button represents the default user response. Any text is displayed within the button. Windows sends a message to the parent window when the user clicks the button.
BS_GROUPBOX	Designates a rectangle into which other buttons are grouped. Any text is displayed in the rectangle's upper-left corner.
BS_LEFTTEXT	Causes text to appear on the left side of the radio button or check-box button. Use this style with the BS_CHECKBOX, BS_RADIOBUTTON, or BS_3STATE styles.
BS_OWNERDRAW	Designates an owner-draw button. The parent window is notified when the button is clicked. Notification includes a request to paint, invert, and disable the button.
BS_PUSHBUTTON	Designates a button that contains the given text. The control sends a message to its parent window whenever the user clicks the button.
BS_RADIOBUTTON	Designates a small circular button that can be checked; its border is bold when the user clicks the button. Any text appears to the right of the button. Typically, two or more radio buttons are grouped together to represent mutually exclusive choices, so no more than one button in the group is checked at any time.
BS_3STATE	Identical to BS_CHECKBOX, except that the box can be grayed as well as checked. The grayed state typically is used to show that a check box has been disabled.

BN_CLICKED

This code specifies that the user has clicked a button. The parent window receives the code through a WM_COMMAND message from a button control.

Parameters

wParam Specifies the control ID.

lParam Contains a handle that identifies the button control in its low-order word and the BN_CLICKED notification code in its high-order word.

Comments Disabled buttons will not send a BN_CLICKED notification message to a parent window.

BN_DOUBLECLICKED

This code specifies that the user has double-clicked a button. The control's parent window receives this code through a WM_COMMAND message from a button control.

Parameters *wParam* Specifies the control ID.

lParam Contains a handle that identifies the button control in its low-order word and the BN_DOUBLECLICKED notification code in its high-order word.

Comments This code applies to buttons with the BS_RADIOBUTTON and BS_OWNERDRAW styles only.

CB_ADDSTRING 3.0

This message adds a string to the list box of a combo box. If the list box is not sorted, the string is added to the end of the list. If the list box is sorted, the string is inserted into the list after sorting.

This message removes any existing list-box selections.

Parameters *wParam* Is not used.

lParam Points to the null-terminated string that is to be added. If the combo box was created with an owner-draw style but without the CBS_HASSTRINGS style, the *lParam* parameter is an application-supplied 32-bit value that is stored by the combo box instead of the pointer to the string.

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

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

CB_DELETESTRING 3.0

This message deletes a string from the list box.

Parameters *wParam* Contains an index to the string that is to be deleted.

lParam Is not used.

Return value The return value is a count of the strings remaining in the list. The return value is CB_ERR if *wParam* does not specify a valid index.

Comments If the combo box was created with an owner-draw style but without the CBS_HASSTRINGS style, a WM_DELETEITEM message is sent to the owner of the combo box so the application can free additional data associated with the item (through the *lParam* parameter of the CB_ADDSTRING or CB_INSERTSTRING message).

CB_DIR 3.0

This message adds a list of the files from the current directory to the list box. Only files with the attributes specified by the *wParam* parameter and that match the file specification given by the *lParam* parameter are added.

Parameters

wParam Contains a DOS attribute value. For a list of the DOS attributes, see the **DlgDirList** function in Chapter 4, "Functions directory."

lParam Points to a file-specification string. The string can contain wildcard characters (for example, *.*).

Return value The return value is a count of items displayed. The return value is CB_ERR if an error occurs; the return value is CB_ERRSPACE if insufficient space is available to store the new strings.

Comments The return value of the CB_DIR message is one less than the return value of the CB_GETCOUNT message.

CB_FINDSTRING 3.0

This message finds the first string in the list box of a combo box which matches the given prefix text.

Parameters

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

lParam Points to the prefix string. The string must be null-terminated.

Return value The return value is the index of the matching item or CB_ERR if the search was unsuccessful.

Comments If the combo box was created with an owner-draw style but without the CBS_HASSTRINGS style, this message returns the index of the item whose long value (supplied as the *lParam* parameter of the CB_ADDSTRING or CB_INSERTSTRING message) matches the value supplied as the *lParam* parameter of CB_FINDSTRING.

CB_GETCOUNT 3.0

This message returns a count of the items in a list box of a combo box.

Parameters

wParam Is not used.

lParam Is not used.

Return value The return value is a count of the items in the list box of a combo box.

CB_GETCURSEL 3.0

This message returns the index of the currently selected item, if any, in the list box of a combo box.

Parameters

wParam Is not used.

lParam Is not used.

Return value The return value is the index of the currently selected item. It is CB_ERR if no item is selected.

CB_GETEDITSEL 3.0

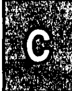
This message returns the starting and ending positions of the selected text in the edit control of a combo box.

Parameters

wParam Is not used.

lParam Is not used.

Return value The return value is a long integer containing the starting position in the low-order word and the ending position in the high-order word. If this message is sent to a combo box without an edit control, the return value is CB_ERR.



CB_GETITEMDATA 3.0

This message retrieves the application-supplied 32-bit value associated with the specified combo-box item. If the item is in an owner-draw combo box created without the CBS_HASSTRINGS style, this 32-bit value was contained in the *lParam* parameter of the CB_ADDSTRING or CB_INSERTSTRING message that added the item to the combo box. Otherwise, it was the value in the *lParam* parameter of a CB_SETITEMDATA message.

Parameters

wParam Contains an index to the item.

lParam Is not used.

Return value The return value is the 32-bit value associated with the item, or CB_ERR if an error occurs.

CB_GETLBTEXT 3.0

This message copies a string from the list box of a combo box into a buffer.

Parameters

wParam Contains the index of the string to be copied.

lParam Points to a buffer that is to receive the string. The buffer must have sufficient space for the string and a terminating null character.

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

Comments If the combo box was created with an owner-draw style but without the CBS_HASSTRINGS style, the buffer pointed to by the *lParam* parameter of the message receives the 32-bit value associated with the item through the *lParam* parameter of the CB_ADDSTRING or CB_INSERTSTRING message.

CB_GETLBTEXTLEN 3.0

This message returns the length of a string in the list box of a combo box.

Parameters

wParam Contains the index of the string.

lParam Is not used.

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

CB_INSERTSTRING 3.0

This message inserts a string into the list box of a combo box. No sorting is performed.

Parameters

wParam Contains an index to the position that will receive the string. If the *wParam* parameter is –1, the string is added to the end of the list.

lParam Points to the null-terminated string that is to be inserted. If the combo box was created with an owner-draw style but without the CBS_HASSTRINGS style, the *lParam* parameter is an application-supplied 32-bit value that is stored by the combo box instead of the pointer to the string.

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

CB_LIMITTEXT 3.0

This message limits the length (in bytes) of the text that the user may enter into the edit control of a combo box.

Parameters

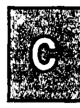
wParam Specifies the maximum number of bytes which the user can enter.

lParam Is not used.

Return value The return value is TRUE if the message is successful; otherwise, it is FALSE. If this message is sent to a combo box without an edit control, the return value is CB_ERR.

CB_RESETCONTENT 3.0

This message removes all strings from the list box of a combo box and frees any memory allocated for those strings.



Parameters *wParam* Is not used.

lParam Is not used.

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

CB_SELECTSTRING 3.0

This message selects the first string in the list box of a combo box that matches the specified prefix. The text in the edit control of the combo box is changed to reflect the new selection.

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

lParam Points to the prefix string. The string must have a null-terminating character.

Return value The return value is the index of the newly selected item. If the search was unsuccessful, the return value is CB_ERR and the current selection is not changed.

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

If the combo box was created with an owner-draw style but without the CBS_HASSTRINGS style, this message returns the index of the item whose long value (supplied as the *lParam* parameter of the CB_ADDSTRING or CB_INSERTSTRING message) matches the value supplied as the *lParam* parameter of CB_FINDSTRING.

CB_SETCURSEL 3.0

This message selects a string in the list box of a combo box and scrolls it into view if the list box is visible, and the text in the combo-box edit control or static-text control is changed to reflect the new selection. When the new string is selected, the list box removes the highlight from the previously selected string.

Parameters *wParam* Contains the index of the string that is to be selected. If *wParam* is –1, the list box is set to have no selection.

lParam Is not used.

Return value If the index specified by *wParam* is not valid, the return value is CB_ERR and the current selection is not changed.

CB_SETEDITSEL 3.0

This message selects all characters in the edit control of a combo box that are within the starting and ending character positions specified by the *lParam* parameter.

Parameters *wParam* Is not used.

lParam Specifies the starting position in the low-order word and the ending position in the high-order word.

Return value The return value is TRUE if the message is successful; otherwise, it is FALSE. If this message is sent to a combo box without an edit control, the return value is CB_ERR.

CB_SETITEMDATA 3.0

This message sets the 32-bit value associated with the specified item in a combo box. If the item is in an owner-draw combo box created without the CBS_HASSTRINGS style, this message replaces the 32-bit value that was contained in the *lParam* parameter of the CB_ADDSTRING or CB_INSERTSTRING message that added the item to the combo box.

Parameters *wParam* Contains an index to the item.

lParam Contains the new value to be associated with the item.

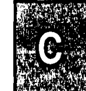
Return value The return value is CB_ERR if an error occurs.

CB_SHOWDROPDOWN 3.0

This message shows or hides the drop-down list box on a combo box created with the CBS_DROPDOWN or CBS_DROPDOWNLIST style.

Parameters *wParam* If TRUE, displays the list box if it is not already visible. If FALSE, hides the list box if it is visible.

lParam Not used.



CBN_DBLCLK 3.0

This code specifies that the user has double-clicked a string in the list box of a combo box. The control's parent window receives this code through a WM_COMMAND message from the control.

Parameters

wParam Specifies the control ID of the combo box.

lParam Contains the combo-box window handle in its low-order word and the CBN_DBLCLK code in its high-order word.

Comments This message can only occur for a combo box with a list box that is always visible. For combo boxes with drop-down list boxes, a single closes the list box and so a double-click cannot occur.

CBN_DROPDOWN 3.0

This code specifies that the list box of a combo box will be dropped down. It is sent just before the combo-box list box is made visible. The control's parent window receives this code through a WM_COMMAND message from the control.

Parameters

wParam Specifies the control ID of the combo box.

lParam Contains the combo-box window handle in its low-order word and the CBN_DROPDOWN code in the high-order word.

Comments This message does not occur if the combo box does not contain a drop-down list box.

CBN_EDITCHANGE 3.0

This code indicates that the user has taken an action that may have altered the text in the edit control of a combo box. It is sent after Windows updates the display (unlike the CBN_EDITUPDATE code). The control's parent window receives this code through a WM_COMMAND message from the control.

Parameters

wParam Specifies the control ID of the combo box.

lParam Contains the combo-box window handle in its low-order word and the CBN_EDITCHANGE code in its high-order word.

Comments This message does not occur if the combo box does not contain an edit control.

CBN_EDITUPDATE 3.0

This code specifies that a combo box containing an edit control will display altered text. The control's parent window receives this code through a WM_COMMAND message from the control.

Parameters

wParam Specifies the control ID of the combo box.

lParam Contains the combo-box window handle in its low-order word and the CBN_EDITUPDATE code in its high-order word.

Comments This message does not occur if the combo box does not contain an edit control.

CBN_ERRSPACE 3.0

This code specifies that the combo-box list-box control cannot allocate enough memory to meet a specific request. The control's parent window receives this code through a WM_COMMAND message from the control.

Parameters

wParam Specifies the control ID of the combo box.

lParam Contains the combo-box window handle in its low-order word and the CBN_ERRSPACE code in its high-order word.

CBN_KILLFOCUS 3.0

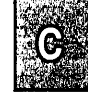
This code is sent when a combo box loses input focus. The control's parent window receives this code through a WM_COMMAND message from the control.

Parameters

wParam Specifies the control ID of the combo box.

lParam Contains the combo-box window handle in its low-order word and the CBN_KILLFOCUS code in its high-order word.

CBN_SELCHANGE 3.0



This code indicates that the selection in the list box of a combo box has changed either as a result of the user clicking in the list box or entering text in the edit control. The control's parent window receives this code through a WM_COMMAND message from the control.

Parameters

wParam Specifies the control ID of the combo box.

lParam Contains the combo-box window handle in its low-order word and the CBN_SELCHANGE code in its high-order word.

CBN_SETFOCUS 3.0

This code is sent when the combo box receives input focus. The control's parent window receives this code through a WM_COMMAND message from the control.

Parameters

wParam Specifies the control ID of the combo box.

lParam Contains the combo-box window handle in its low-order word and the CBN_SETFOCUS code in its high-order word.

DM_GETDEFID

This message retrieves the ID of the default push-button control for a dialog box.

Parameters

wParam Is not used.

lParam Is not used.

Return value The return value is a 32-bit value. The high-order word contains DC_HASDEFID if the default button exists; otherwise, it is NULL. The low-order word contains the ID of the default button if the high-order word contains DC_HASDEFID; otherwise, it is zero.

DM_SETDEFID

This message is used by an application to change the default push-button control ID for a dialog box.

Parameters

wParam Contains the ID of the new default push-button control.

lParam Is not used.

EM_CANUNDO

This message determines whether an edit control can respond correctly to an EM_UNDO message.

Parameters *wParam* Is not used.

lParam Is not used.

Return value The return value is nonzero if the edit control can process the EM_UNDO message correctly. Otherwise, it is zero.

EM_EMPTYUNDOBUFFER 3.0

This message directs an edit control to clear its undo buffer. This disables the edit control's ability to undo the last edit.

Parameters *wParam* Is not used.

lParam Is not used.

Comments The undo buffer is automatically emptied whenever the edit control receives a WM_SETTEXT or EM_SETHANDLE message.

EM_FMTLINES

This message directs a multiline edit control to add or remove the end-of-line character from word wrapped text lines.

Parameters *wParam* Indicates the disposition of end-of-line characters. If the *wParam* parameter is nonzero, the characters CR CR LF (0D 0D 0A hexadecimal) are placed at the end of wordwrapped lines. If *wParam* is zero, the end-of-line characters are removed from the text.

lParam Is not used.

Return value The return value is nonzero if any formatting occurs. Otherwise, it is zero.

Comments Lines that end with a hard return (a carriage return entered by the user) contain the characters CR LF at the end of the line. These lines are not affected by the EM_FMTLINES message.

Notice that the size of the text changes when this message is processed.

EM_GETHANDLE

This message returns the data handle of the buffer that holds the contents of the control window. The handle is always a local handle to a location in the application's data segment.

Parameters *wParam* Is not used.

lParam Is not used.

Return value The return value is a data handle that identifies the buffer that holds the contents of the edit control.

Comments An application may send this message to a control only if it has created the dialog box containing the control with the DS_LOCALEDIT style flag set.

EM_GETLINE

This message copies a line from the edit control.

Parameters *wParam* Specifies the line number of the line in the control, where the line number of the first line is zero.

lParam Points to the buffer where the line will be stored. The first word of the buffer specifies the maximum number of bytes to be copied to the buffer. The copied line is not null-terminated.

Return value The return value is the number of bytes actually copied. This message is not processed by single-line edit controls.

EM_GETLINECOUNT

This message returns the number of lines of text in the edit control.

Parameters *wParam* Is not used.

lParam Is not used.

Return value The return value is the number of lines of text in the control.

Comments This message is not processed by single-line edit controls.

EM_GETMODIFY

This message returns the current value of the modify flag for a given edit control. The flag is set by the control if the user enters or modifies text within the control.

Parameters

wParam Is not used.

lParam Is not used.

Return value The return value is the value of the current modify flag for a given edit control.

EM_GETRECT

This message retrieves the formatting rectangle of the control.

Parameters

wParam Is not used.

lParam Points to a **RECT** data structure. The control copies the dimensions of the structure.

EM_GETSEL

This message returns the starting and ending character positions of the current selection.

Parameters

wParam Is not used.

lParam Is not used.

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

EM_LIMITTEXT

This message limits the length (in bytes) of the text the user may enter.

Parameters

wParam Specifies the maximum number of bytes that can be entered. If the user attempts to enter more characters, the edit control beeps and does not accept the characters. If the *wParam* parameter is zero, no limit is imposed on the size of the text (until no more memory is available).

	lParam	Is not used.

Comments The EM_LIMITTEXT message does not affect text set by the WM_SETTEXT message or the buffer set by the EM_SETHANDLE message.

EM_LINEFROMCHAR

This message returns the line number of the line that contains the character whose position (indexed from the beginning of the text) is specified by the *wParam* parameter.

Parameters

wParam	Contains the index value for the desired character in the text of the edit control (these index values are zero-based), or contains –1.
lParam	Is not used.

Return value The return value is a line number. If *wParam* is –1, the number of the line that contains the first character of the selection is returned; otherwise, *wParam* contains the index (or position) of the desired character in the edit-control text, and the number of the line that contains that character is returned.

EM_LINEINDEX

This message returns the number of character positions that occur preceding the first character in a given line.

Parameters

wParam	Specifies the desired line number, where the line number of the first line is zero. If the *wParam* parameter is –1, the current line number (the line that contains the caret) is used.
lParam	Is not used.

Return value The return value is the number of character positions that precede the first character in the line.

Comments This message will not be processed by single-line edit controls.

EM_LINELENGTH

This message returns the length of a line (in bytes) in the edit control's text buffer.

Parameters *wParam* Specifies the character index of a character in the specified line, where the line number of the first line is zero. If the *wParam* parameter is –1, the length of the current line (the line that contains the caret) is returned, not including the length of any selected text. If the current selection spans more than one line, the total length of the lines, minus the length of the selected text, is returned.

lParam Is not used.

Comments Use the EM_LINEINDEX message to retrieve a character index for a given line number. This index can be used with the EM_LINELENGTH message.

EM_LINESCROLL

This message scrolls the content of the control by the given number of lines.

Parameters *wParam* Is not used.

lParam Contains the number of lines and character positions to scroll. The low-order word of the *lParam* parameter contains the number of lines to scroll vertically; the high-order word contains the number of character positions to scroll horizontally.

Comments This message will not be processed by single-line edit controls.

EM_REPLACESEL

This message replaces the current selection with new text.

Parameters *wParam* Is not used.

lParam Points to a null-terminated string of replacement text.

EM_SETHANDLE

This message establishes the text buffer used to hold the contents of the control window.

Parameters *wParam* Contains a handle to the buffer. The handle must be a local handle to a location in the application's data segment. The edit control uses this buffer to store the currently displayed text, instead of allocating its own buffer. If necessary, the control reallocates this buffer.

lParam Is not used.

Comments This message will not be processed by single-line edit controls.

If the EM_SETHANDLE message is used to change the text buffer used by an edit control, the previous text buffer is not destroyed. The application must retrieve the previous buffer handle before setting the new handle, and must free the old handle by using the **LocalFree** function.

An edit control automatically reallocates the given buffer whenever it needs additional space for text, or it removes enough text so that additional space is no longer needed. An application may send this message to a control only if it has created the dialog box containing the control with the DS_LOCALEDIT style flag set.

EM_SETMODIFY

This message sets the modify flag for a given edit control.

Parameters *wParam* Specifies the new value for the modify flag.

lParam Is not used.

EM_SETPASSWORDCHAR 3.0

This message sets the character displayed in an edit control created with the ES_PASSWORD style. The default display character is an asterisk (*).

Parameters *wParam* Specifies the character to be displayed in place of the character typed by the user. If *wParam* is NULL, the actual characters typed by the user are displayed.

lParam Is not used.

EM_SETRECT

This message sets the formatting rectangle for a control. The text is reformatted and redisplayed to reflect the changed rectangle.

Parameters
wParam Is not used.

lParam Points to a **RECT** data structure that specifies the new dimensions of the rectangle.

Comments This message will not be processed by single-line edit controls.

EM_SETRECTNP

This message sets the formatting rectangle for a control. The text is reformatted and redisplayed to reflect the changed rectangle. The EM_SETRECTNP message is the same as the EM_SETRECT message, except that the control is *not* repainted. Any subsequent alterations cause the control to be repainted to reflect the changed formatting rectangle. This message is used when the field is to be repainted later.

Parameters
wParam Is not used.

lParam Points to a **RECT** data structure that specifies the new dimensions of the rectangle.

Comments This message will not be processed by single-line edit controls.

EM_SETSEL

This message selects all characters in the current text that are within the starting and ending character positions given by the *lParam* parameter.

Parameters
wParam Is not used.

lParam Specifies the starting position in the low-order word and the ending position in the high-order word. The position values 0 to 32,767 select the entire string.

EM_SETTABSTOPS 3.0

This message sets the tab-stop positions in a multiline edit control.

Parameters
wParam Is an integer that specifies the number of tab stops in the edit control.

lParam Is a long pointer to the first member of an array of integers containing the tab stop positions in dialog units. (A dialog unit is a horizontal or vertical distance. One horizontal dialog unit

is equal to 1/4 of the current dialog base width unit. The dialog base units are computed based on the height and width of the current system font. The **GetDialogBaseUnits** function returns the current dialog base units in pixels.) The tab stops must be sorted in increasing order; back tabs are not allowed.

Return value The return value is TRUE if all the tabs were set. Otherwise, the return value is FALSE.

Comments If *wParam* is zero and *lParam* is NULL, the default tab stops are set at every 32 dialog units. If *wParam* is 1, the edit control will have tab stops separated by the distance specified by *lParam*. If *lParam* points to more than a single value, then a tab stop will be set for each value in *lParam*, up to the number specified by *wParam*.

EM_SETWORDBREAK

This message is sent to the multiline edit control, informing the edit control that Windows has replaced the default word-break function with an application-supplied word-break function. A word-break function scans a text buffer (which contains text to be sent to the display), looking for the first word that will not fit on the current display line. The word-break function places this word at the beginning of the next line on the display. A word-break function defines at what point Windows should break a line of text for multiline edit controls, usually at a blank character that separates two words. The default word-break function breaks a line of text at a blank character. The application-supplied function may define a word break to be a hyphen or character other than the blank character.

Parameters

wParam Is not used.

lParam Is a procedure-instance address.

Comments The callback-function address, passed as the *lParam* parameter, must be created by using the **MakeProcInstance** function. The callback function must use the Pascal calling convention and must be declared **FAR**.

Callback Function

LPSTR FAR PASCAL *WordBreakFunc(lpchEditText, ichCurrentWord, cchEditText)*
LPSTR *lpchEditText;*
short *ichCurrentWord;*
short *cchEditText;*

WordBreakFunc is a placeholder for the application-supplied function name. The actual name must be exported by including it in an **EXPORTS** statement in the application's module-definition file.

Parameters

lpchEditText	Points to the text of the edit control.
ichCurrentWord	Specifies an index to a word in the buffer of text that identifies at what point the function should begin checking for a word break.
cchEditText	Specifies the number of bytes of edit text.

Return value The return value points to the first byte of the next word in the edit-control text. If the current word is the last word in the text, the return value points to the first byte that follows the last word.

EM_UNDO

This message undoes the last edit to the edit control. When the user modifies the edit control, the last change is stored in an undo buffer, which grows dynamically as required. If insufficient space is available for the buffer, the undo attempt fails and the edit control is unchanged.

Parameters

wParam	Is not used.
lParam	Is not used.

Return value The return value is nonzero if the undo operation is successful. It is zero if the undo operation fails.

EN_CHANGE

This code specifies that the user has taken an action that may have altered text. It is sent after Windows updates a display (unlike the EN_UPDATE code). The control's parent window receives this code through a WM_COMMAND message from the control.

Parameters

wParam	Contains the *wParam* parameter of the WM_COMMAND message, and specifies the control ID.
lParam	Contains an edit-control window handle in its low-order word and the EN_CHANGE code in its high-order word.

EN_ERRSPACE

This code specifies that the edit control cannot allocate additional memory space. The control's parent window receives this code through a WM_COMMAND message from the control.

Parameters

wParam Contains the *wParam* parameter of the WM_COMMAND message, and specifies the control ID.

lParam Contains an edit-control window handle in its low-order word and the EN_ERRSPACE code in its high-order word.

EN_HSCROLL

This code specifies that the user has clicked the edit control's horizontal scroll bar. The control's parent window receives this code through a WM_COMMAND message from the control. The parent window is notified before the screen is updated.

Parameters

wParam Contains the *wParam* parameter of the WM_COMMAND message, and specifies the control ID.

lParam Contains an edit-control window handle in its low-order word and the EN_HSCROLL code in its high-order word.

EN_KILLFOCUS

This code specifies that the edit control has lost the input focus. The control's parent window receives this code through a WM_COMMAND message from the control.

Parameters

wParam Contains the *wParam* parameter of the WM_COMMAND message, and specifies the control ID.

lParam Contains an edit-control window handle in its low-order word and the EN_KILLFOCUS code in its high-order word.

EN_MAXTEXT 3.0

This code specifies that the current insertion has exceeded the specified number of characters for the edit control. The insertion has been truncated. This message is also sent when an edit control does not have the ES_AUTOHSCROLL style and the number of characters to be inserted would exceed the width of the edit control. The control's parent window receives this code through a WM_COMMAND message from the control.

Parameters

wParam Contains the *wParam* parameter of the WM_COMMAND message, and specifies the control ID.

lParam Contains an edit-control window handle in its low-order word and the EN_MAXTEXT code in its high-order word.

EN_SETFOCUS

This code specifies that the edit control has obtained the input focus. The control's parent window receives this code through a WM_COMMAND message from the control.

Parameters

wParam Contains the *wParam* parameter of the WM_COMMAND message, and specifies the control ID.

lParam Contains an edit-control window handle in its low-order word and the EN_SETFOCUS code in its high-order word.

EN_UPDATE

The code specifies that the edit control will display altered text. The control's parent window receives this code through a WM_COMMAND message from the control; notification occurs after the control has formatted the text, but before it displays the text. This makes it possible to alter the window size, if necessary.

Parameters

wParam Specifies the control ID.

lParam Contains an edit-control window handle in its low-order word and the EN_UPDATE code in its high-order word.

EN_VSCROLL

This code specifies that the user has clicked the edit control's vertical scroll bar. The control's parent window receives this code through a WM_COMMAND message from the control; notification occurs before the screen is updated.

Parameters *wParam* Contains the *wParam* parameter of the WM_COMMAND message, and specifies the control ID.

lParam Contains an edit-control window handle in its low-order word and the EN_VSCROLL code in its high-order word.

LB_ADDSTRING

This message adds a string to the list box. If the list box is not sorted, the string is added to the end of the list. If the list box is sorted, the string is inserted into the list after sorting.

This message removes any existing list-box selections.

Parameters *wParam* Is not used.

lParam Points to the null-terminated string that is to be added. If the list box was created with an owner-draw style but without the LBS_HASSTRINGS style, the *lParam* parameter is an application-supplied 32-bit value that is stored by the list box instead of the pointer to the string.

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

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

LB_DELETESTRING

This message deletes a string from the list box.

Parameters *wParam* Contains an index to the string that is to be deleted.

lParam Is not used.

Return value The return value is a count of the strings remaining in the list. The return value is LB_ERR if an error occurs.

Comments If the list box was created with an owner-draw style but without the LBS_HASSTRINGS style, a WM_DELETEITEM message is sent to the owner of the list box so the application can free additional data associated with the item (through the *lParam* parameter of the LB_ADDSTRING or LB_INSERTSTRING message).

LB_DIR

This message adds a list of the files from the current directory to the list box. Only files with the attributes specified by the *wParam* parameter and that match the file specification given by the *lParam* parameter are added.

Parameters

wParam Contains a DOS attribute value. For a list of the DOS attributes, see the **DlgDirList** function in Chapter 4, "Functions directory."

lParam Points to a file-specification string. The string can contain wildcard characters (for example, *.*).

Return value The return value is a count of items displayed. The return value is LB_ERR if an error occurs; the return value is LB_ERRSPACE if insufficient space is available to store the new strings.

Comments The return value of the LB_DIR message is one less than the return value of the LB_GETCOUNT message.

LB_FINDSTRING 3.0

This message finds the first string in the list box which matches the given prefix text.

Parameters

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

lParam Points to the prefix string. The string must be null-terminated.

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

Comments If the list box was created with an owner-draw style but without the LBS_HASSTRINGS style, this message returns the index of the item whose long value (supplied as the *lParam* parameter of the LB_ADDSTRING or LB_INSERTSTRING message) matches the value supplied as the *lParam* parameter of LB_FINDSTRING.

LB_GETCARETINDEX 3.1

This message returns the index of the item that has the focus caret in a list box. If the list box is a single-selection list box, the item will also be selected. In a multiple-selection list box, the item is not necessarily a selected item.

Parameters *wParam* Is not used.

lParam Is not used.

Return value The return value is the list-box item that has the focus caret. The return value is LB_ERR if an error occurs.

LB_GETCOUNT

This message returns a count of the items in the list box.

Parameters *wParam* Is not used.

lParam Is not used.

Return value The return value is a count of the items in the list box. The return value is LB_ERR if an error occurs.

LB_GETCURSEL

This message returns the index of the currently selected item, if any.

Parameters *wParam* Is not used.

lParam Is not used.

Return value The return value is the index of the currently selected item. It is LB_ERR if no item is selected or if the list-box type is multiple selection.

LB_GETHORIZONTALEXTENT 3.0

This message retrieves from a list box the width in pixels by which the list box can be scrolled horizontally if the list box has horizontal scroll bars.

Parameters *wParam* Is not used.

lParam Is not used.

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

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

LB_GETITEMDATA 3.0

This message retrieves the application-supplied 32-bit value associated with the specified list-box item. If the item is in an owner-draw list box created without the LBS_HASSTRINGS style, this 32-bit value was contained in the *lParam* parameter of the LB_ADDSTRING or LB_INSERTSTRING message that added the item to the list box. Otherwise, it was the value in the *lParam* parameter of a LB_SETITEMDATA message.

Parameters *wParam* Contains an index to the item.

lParam Is not used.

Return value The return value is the 32-bit value associated with the item, or LB_ERR if an error occurs.

LB_GETITEMHEIGHT 3.1

This message returns the height of one or all items in the list box. If the list box is a variable-height owner-draw list box, this message returns the height of the item specified by the *wParam* parameter. Otherwise, this message returns the height of all items in the list box.

Parameters *wParam* Specifies the index of the item for which the height is to be obtained.

lParam Is not used.

Return value The return value specifies the height in pixels of the item.

The return value is LB_ERR if there is an error.

LB_GETITEMRECT 3.0

This message retrieves the dimensions of the rectangle that bounds a list-box item as it is currently displayed in the list-box window.

Parameters *wParam* Contains an index to the item.

lParam Contains a long pointer to a **RECT** data structure that receives the list-box client coordinates of the item.

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

LB_GETSEL

This message returns the selection state of an item.

Parameters *wParam* Contains an index to the item.

lParam Is not used.

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

LB_GETSELCOUNT 3.0

This message returns the total number of selected items in a multiselection list box.

Parameters *wParam* Not used.

lParam Not used.

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

LB_GETSELITEMS 3.0

This message fills a buffer with an array of integers specifying the item numbers of selected items in a multiselection list box.

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

	lParam	Contains a long pointer to a buffer large enough for the number of integers specified by the *wParam* parameter.
Return value	The return value is the actual number of items placed in the buffer. If the list box is a single-selection list box, the return value is LB_ERR.	

LB_GETTEXT

This message copies a string from the list into a buffer.

Parameters	*wParam*	Contains the index of the string to be copied.
	lParam	Points to the buffer that is to receive the string. The buffer must have both sufficient space for the string and a terminating null character.

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

Comments If the list box was created with an owner-draw style but without the LBS_HASSTRINGS style, the buffer pointed to by the *lParam* parameter of the message receives the 32-bit value associated with the item through the *lParam* parameter of the LB_ADDSTRING or LB_INSERTSTRING message.

LB_GETTEXTLEN

This message returns the length of a string in the list box.

Parameters	*wParam*	Contains an index to the string.
	lParam	Is not used.

Return value The return value is the length of the string (in bytes), excluding the terminating null character. The return value is LB_ERR if an error occurs.

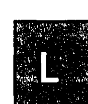
LB_GETTOPINDEX 3.0

This message returns the index of the first visible item in a list box. Initially, item 0 is at the top of the list box, but if the list box is scrolled, another item may be at the top.

Parameters	*wParam*	Not used.

	lParam	Not used.
Return value	The index of the first visible item in a list box.	

LB_INSERTSTRING



This message inserts a string into the list box. No sorting is performed.

Parameters

wParam Contains an index to the position that will receive the string. If the *wParam* parameter is –1, the string is added to the end of the list.

lParam Points to the null-terminated string that is to be inserted. If the list box was created with an owner-draw style but without the LBS_HASSTRINGS style, the *lParam* parameter is an application-supplied 32-bit value that is stored by the list box instead of the pointer to the string.

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

LB_RESETCONTENT

This message removes all strings from a list box and frees any memory allocated for those strings.

Parameters

wParam Is not used.

lParam Is not used.

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

LB_SELECTSTRING

This message changes the current selection to the first string that has the specified prefix.

Parameters

wParam Contains the index of the item before the first item to be searched. When the search reaches the bottom of the list box it continues from the top of the list box back to the item specified

by *wParam*. If the *wParam* parameter is –1, the entire list box is searched from the beginning.

lParam Points to the prefix string. The string must have a null-terminating character.

Return value The return value is the index of the selected item. The return value is LB_ERR if an error occurs.

Comments This message must not be used with list boxes that are multiple-selection type.

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

If the list box was created with an owner-draw style but without the LBS_HASSTRINGS style, this message returns the index of the item whose long value (supplied as the *lParam* parameter of the LB_ADDSTRING or LB_INSERTSTRING message) matches the value supplied as the *lParam* parameter of LB_FINDSTRING.

LB_SELITEMRANGE 3.0

This message selects one or more consecutive items in a multiple-selection list box.

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

lParam The low-order word of the *lParam* parameter is an index that specifies the first item to set, and the high-order word is an index that specifies the last item to set.

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

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

LB_SETCARETINDEX 3.1

This message is sent to a multiple-selection list box to set the focus caret on an item. If the item is not visible, it is scrolled into view.

Parameters *wParam* Specifies the index of the item to receive focus.

	lParam	Is not used.

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

Comments This message must be used with list boxes that are multiple-selection type only.

LB_SETCOLUMNWIDTH 3.0

This message is sent to a multicolumn list box created with the LBS_MULTICOLUMN style to set the width in pixels of all columns in the list box.

Parameters

wParam	Specifies the width in pixels of all columns.
lParam	Is not used.

LB_SETCURSEL

This message selects a string and scrolls it into view, if necessary. When the new string is selected, the list box removes the highlight from the previously selected string.

Parameters

wParam	Contains the index of the string that is selected. If *wParam* is –1, the list box is set to have no selection.
lParam	Is not used.

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

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

LB_SETHORIZONTALEXTENT 3.0

This message sets the width in pixels by which a list box can be scrolled horizontally. If the size of the list box is smaller than this value, the horizontal scroll bar will horizontally scroll items in the list box. If the list box is as large or larger than this value, the horizontal scroll bar is disabled.

Parameters

wParam	Specifies the number of pixels by which the list box can be scrolled.
lParam	Is not used.

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

LB_SETITEMDATA 3.0

This message sets a 32-bit value associated with the specified item in a list box. If the item is in an owner-draw list box created without the LBS_HASSTRINGS style, this message replaces the 32-bit value that was contained in the *lParam* parameter of the LB_ADDSTRING or LB_INSERTSTRING message that added the item to the list box.

Parameters

wParam Contains an index to the item.

lParam Contains the new value to be associated with the item.

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

LB_SETITEMHEIGHT 3.1

This message is sent to a list box to set the height of one or all items in the list box. If the list box is a variable-height owner-draw list box, this message sets the height of the item specified by the *wParam* parameter. Otherwise, this message sets the height of all items in the list box.

Parameters

wParam Specifies the index of the item for which the height is to be set.

lParam Specifies the new height in pixels of the item.

Return value The return value is LB_ERR if *wParam* does not specify a valid item index or if *lParam* specifies an invalid height.

LB_SETSEL

This message selects a string in a multiple-selection list box.

Parameters

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

lParam The low-order word of the *lParam* parameter is an index that specifies which string to set. If *lParam* is –1, the selection is

added to or removed from all strings, depending on the value of *wParam*.

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

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

LB_SETTABSTOPS 3.0

This message sets the tab-stop positions in a list box.

Parameters

wParam Is an integer that specifies the number of tab stops in the list box.

lParam Is a long pointer to the first member of an array of integers containing the tab stop positions in dialog units. (A dialog unit is a horizontal or vertical distance. One horizontal dialog unit is equal to 1/4 of the the current dialog base width unit. The dialog base units are computed based on the height and width of the current system font. The **GetDialogBaseUnits** function returns the current dialog base units in pixels.) The tab stops must be sorted in increasing order; back tabs are not allowed.

Return value The return value is TRUE if all the tabs were set. Otherwise, the return value is FALSE.

Comments If *wParam* is zero and *lParam* is NULL, the default tab stop is two dialog units.

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

If *lParam* points to more than a single value, then a tab stop will be set for each value in *lParam*, up to the number specified by *wParam*.
To respond to the LB_SETTABSTOPS message, the list box must have been created with the LBS_USETABSTOPS style.

LB_SETTOPINDEX 3.0

This message sets the first visible item in a list box to the item identified by the index.

Parameters

wParam Specifies the index of the list-box item.

lParam Not used.

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

LBN_DBLCLK

This code specifies that the user has double-clicked a string. The control's parent window receives this code through a WM_COMMAND message from the control.

Parameters

wParam	Contains the *wParam* parameter of the WM_COMMAND message, and specifies the control ID.
lParam	Contains an edit-control window handle in its low-order word and the LBN_DBLCLK code in its high-order word.

Comments This code applies only to list-box controls that have LBS_NOTIFY style.

LBN_ERRSPACE

This code specifies that the list-box control cannot allocate enough memory to meet a specific request. The control's parent window receives this code through a WM_COMMAND message from the control.

Parameters

wParam	Contains the *wParam* parameter of the WM_COMMAND message, and specifies the control ID.
lParam	Contains a list-box window handle in its low-order word and the LBN_ERRSPACE code in its high-order word.

Comments This code applies only to list-box controls that have LBS_NOTIFY style.

LBN_KILLFOCUS 3.0

This code is sent when a list box loses input focus. The control's parent window receives this code through a WM_COMMAND message from the control.

Parameters

wParam	Specifies the control ID of the list box.
lParam	Contains the list-box window handle in its low-order word and the LBN_KILLFOCUS code in its high-order word.

LBN_SELCHANGE

This code specifies that the selection in a list box has changed. The control's parent window receives this code through a WM_COMMAND message from the control.

Parameters

wParam Contains the *wParam* parameter of the WM_COMMAND message, and specifies the control ID.

lParam Contains a list-box window handle in its low-order word and the LBN_SELCHANGE code in its high-order word.

Comments This code applies only to list-box controls that have LBS_NOTIFY style.

LBN_SETFOCUS 3.0

This code is sent when the list box receives input focus. The control's parent window receives this code through a WM_COMMAND message from the control.

Parameters

wParam Specifies the control ID of the list box.

lParam Contains the list-box window handle in its low-order word and the LBN_SETFOCUS code in its high-order word.

WM_ACTIVATE

This message is sent when a window becomes active or inactive.

Parameters

wParam Specifies the new state of the window. The *wParam* parameter is zero if the window is inactive; it is one of the following nonzero values if the window is being activated:

Value	Meaning
1	The window is being activated through some method other than a mouse click (for example, through a call to the **Set-ActiveWindow** function or selection of the window by the user through the keyboard interface).
2	The window is being activated by a mouse click by the user. Any mouse button can be clicked: right, left, or middle.

lParam Identifies a window and specifies its state. The high-order word of the *lParam* parameter is nonzero if the window is minimized. Otherwise, it is zero. The value of the low-order word of *lParam* depends on the value of the *wParam* parameter. If *wParam* is zero, the low-order word of *lParam* is a handle to the window being activated. If *wParam* is nonzero, the low-order word of *lParam* is the handle of the window being inactivated (this handle may be NULL).

Default action If the window is being activated and is not minimized, the **DefWindowProc** function sets the input focus to the window.

WM_ACTIVATEAPP

This message is sent when a window being activated belongs to a different application than the currently active window. The message is sent to the application whose window will be activated and the application whose window will be deactivated.

Parameters

wParam Specifies whether a window is being activated or deactivated. A nonzero value indicates that Windows will activate a window; zero indicates that Windows will deactivate a window.

lParam Contains the task handle of the application. If the *wParam* parameter is zero, the low-order word of the *lParam* parameter contains the task handle of the application that owns the window that is being deactivated. If *wParam* is nonzero, the low-order word of *lParam* contains the task handle of the application that owns the window that is being activated. The high-order word is not used.

WM_ASKCBFORMATNAME

This message is sent when the clipboard contains a data handle for the CF_OWNERDISPLAY format (that is, the clipboard owner should display the clipboard contents), and requests a copy of the format name.

Parameters

wParam Specifies the maximum number of bytes to copy.

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

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

WM_CANCELMODE

This message cancels any mode the system is in, such as one that tracks the mouse in a scroll bar or moves a window. Windows sends the WM_CANCELMODE message when an application displays a message box.

Parameters *wParam* Is not used.

lParam Is not used.

WM_CHANGECBCHAIN

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

Parameters *wParam* Contains the handle to the window that is being removed from the clipboard-viewer chain.

lParam Contains in its low-order word the handle to the window that follows the window being removed from the clipboard-viewer chain.

Comments Each window that receives the WM_CHANGECBCHAIN message should call the **SendMessage** function to pass on the message to the next window in the clipboard-viewer chain. If the window being removed is the next window in the chain, the window specified by the low-order word of the *lParam* parameter becomes the next window, and clipboard messages are passed on to it.

WM_CHAR

This message results when a WM_KEYUP and a WM_KEYDOWN message are translated. It contains the value of the keyboard key being pressed or released.

Parameters *wParam* Contains the value of the key.

lParam Contains the repeat count, scan code, key-transition code, previous key state, and context code, as shown in the following list:

Bit	Value
0–15 (low-order word)	Repeat count (the number of times the key stroke is repeated as a result of the user holding down the key).
16–23 (low byte of high-order word)	Scan code (OEM-dependent value).
24	Extended key, such as a function key or a key on the numeric keypad (1 if it is an extended key).
25–26	Not used.
27–28	Used internally by Windows.
29	Context code (1 if the ALT key is held down while the key is pressed, 0 otherwise).
30	Previous key state (1 if the key is down before the message is sent, 0 if the key is up).
31	Transition state (1 if the key is being released, 0 if the key is being pressed).

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

For IBM® Enhanced 101- and 102-key keyboards, enhanced keys are the right ALT and the right CONTROL keys on the main section of the keyboard; the INSERT, DELETE, HOME, END, PAGE UP, PAGE DOWN and DIRECTION keys in the clusters to the left of the numeric key pad; and the divide (/) and ENTER keys in the numeric key pad. Some other keyboards may support the extended-key bit in the *lParam* parameter.

WM_CHARTOITEM 3.0

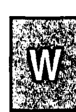
This message is sent by a list box with the LBS_WANTKEYBOARDINPUT style to its owner in response to a WM_CHAR message.

Parameters *wParam* Contains the value of the key which the user pressed.

lParam Contains the current caret position in its high-order word and the window handle of the list box in its low-order word.

Return value The return value specifies the action which the application performed in response to the message. A return value of –2 indicates that the application handled all apsects of selecting the item and wants no further action by the list box. A return value of –1 indicates that the list box should perform the default action in response to the key stroke. A return value of zero or greater specifies the index of an item in the list box and indicates that the list box should perform the default action for the key stroke on the given item.

WM_CHILDACTIVATE



This message is sent to a child window's parent window when the **SetWindowPos** function moves a child window.

Parameters *wParam* Is not used.

lParam Is not used.

WM_CLEAR

This message deletes the current selection.

Parameters *wParam* Is not used.

lParam Is not used.

WM_CLOSE

This message occurs when a window is closed.

Parameters *wParam* Is not used.

lParam Is not used.

Default action The **DefWindowProc** function calls the **DestroyWindow** function to destroy the window.

Comments An application can prompt the user for confirmation, prior to destroying a window, by processing the WM_CLOSE message and calling the **DestroyWindow** function only if the user confirms the choice.

WM_COMMAND

This message occurs when the user selects an item from a menu, when a control passes a message to its parent window, or when an accelerator key stroke is translated.

Parameters

wParam Contains the menu item, the control ID, or the accelerator ID.

lParam Specifies whether the message is from a menu, an accelerator, or a control. The low-order word contains zero if the message is from a menu. The high-order word contains 1 if the message is an accelerator message. If the message is from a control, the high-order word of the *lParam* parameter contains the notification code. The low-order word is the window handle of the control sending the message.

Comments

Accelerator key strokes that are defined to select items from the System menu are translated into WM_SYSCOMMAND messages.

If an accelerator key stroke that corresponds to a menu item occurs when the window that owns the menu is minimized, no WM_COMMAND message is sent. However, if an accelerator key stroke that does not match any of the items on the window's menu or on the System menu occurs, a WM_COMMAND message is sent, even if the window is minimized.

WM_COMPACTING 3.0

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

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

Parameters

wParam Specifies the ratio of CPU time currently spent by Windows compacting memory. For example, 8000h represents 50% of CPU time.

lParam Is not used.

WM_COMPAREITEM 3.0

This message determines the relative position of a new item in a sorted owner-draw combo or list box.

Whenever the application adds a new item, Windows sends this message to the owner of a combo or list box created with the CBS_SORT or LBS_SORT style. The *lParam* parameter of the message is a long pointer to a **COMPAREITEMSTRUCT** data structure that contains the identifiers and application-supplied data for two items in the combo or list box. When the owner receives the message, the owner returns a value indicating which of the items should appear before the other. Typically, Windows sends this message several times until it determines the exact position for the new item.

Parameters

wParam Is not used.

lParam Contains a long pointer to a **COMPAREITEMSTRUCT** data structure that contains the identifiers and application-supplied data for two items in the combo or list box.

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

Value	Meaning
–1	Item 1 sorts before item 2.
0	Item 1 and item 2 sort the same.
1	Item 1 sorts after item 2.

WM_COPY

This message sends the current selection to the clipboard in CF_TEXT format.

Parameters

wParam Is not used.

lParam Is not used.

WM_CREATE

This message informs the window procedure that it can perform any initialization. The **CreateWindow** function sends this message before it returns and before the window is opened.

Parameters

wParam Is not used.

lParam Points to a **CREATESTRUCT** data structure that contains copies of parameters passed to the **CreateWindow** function.

WM_CTLCOLOR

This message is sent to the parent window of a predefined control or message box when the control or message box is about to be drawn. By responding to this message, the parent window can set the text and background colors of the child window by using the display-context handle given in the *wParam* parameter.

Parameters

wParam Contains a handle to the display context for the child window.

lParam The low-order word of the *lParam* parameter contains the handle to the child window. The high-order word is one of the following values, specifying the type of control:

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

Default action The **DefWindowProc** function selects the default system colors.

Comments When processing the WM_CTLCOLOR message, the application must align the origin of the intended brush with the window coordinates by first calling the **UnrealizeObject** function for the brush, and then setting the brush origin to the upper-left corner of the window.

If an application processes the WM_CTLCOLOR message, it must return a handle to the brush that is to be used for painting the control background. Note that failure to return a valid brush handle will place the system in an unstable state.

WM_CUT

This message sends the current selection to the clipboard in CF_TEXT format, and then deletes the selection from the control window.

Parameters *wParam* Is not used.

lParam Is not used.

WM_DEADCHAR

This message results when a WM_KEYUP and a WM_KEYDOWN message are translated. It specifies the character value of a dead key. A dead key is a key, such as the umlaut (double-dot) character, that is combined with other characters to form a composite character. For example, the umlaut-O character consists of the dead key, umlaut, and the O key.

Parameters *wParam* Contains the dead-key character value.

lParam Contains the repeat count, scan code, key-transition code, previous key state, and context code, as shown in the following list:

Bit	Value
0–15 (low-order word)	Repeat count (the number of times the key stroke is repeated as a result of the user holding down the key).
16–23 (low byte of high-order word)	Scan code (OEM-dependent value).
24	Extended key, such as a function key or a key on the numeric keypad (1 if it is an extended key, 0 otherwise).
25–26	Not used.
27–28	Used internally by Windows.
29	Context code (1 if the ALT key is held down while the key is pressed, 0 otherwise).
30	Previous key state (1 if the key is down before the message is sent, 0 if the key is up).
31	Transition state (1 if the key is being released, 0 if the key is being pressed).

Comments The WM_DEADCHAR message typically is used by applications to give the user feedback about each key pressed. For example, an application can display the accent in the current character position without moving the caret.

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

For IBM Enhanced 101- and 102-key keyboards, enhanced keys are the right ALT and the right CONTROL keys on the main section of the keyboard; the INSERT, DELETE, HOME, END, PAGE UP, PAGE DOWN and DIRECTION keys in the clusters to the left of the numeric key pad; and the divide (/) and ENTER keys in the numeric key pad. Some other keyboards may support the extended-key bit in the *lParam* parameter.

WM_DELETEITEM 3.0

This message informs the owner of an owner-draw list box or combo box that a list-box item has been removed. This message is sent when the list box or combo box is destroyed or the item is removed by the LB_DELETESTRING, LB_RESETCONTENT, CB_DELETESTRING or CB_RESETCONTENT message.

Parameters

wParam Not used.

lParam Contains a long pointer to a **DELETEITEMSTRUCT** data structure that contains information about the deleted list-box item.

WM_DESTROY

This message informs the window that it is being destroyed. The **DestroyWindow** function sends the WM_DESTROY message to the window after removing the window from the screen. The WM_DESTROY message is sent to a parent window before any of its child windows are destroyed.

Parameters

wParam Is not used.

lParam Is not used.

Comments If the window being destroyed is part of the clipboard-viewer chain (set by using the **SetClipboardViewer** function), the window must remove itself from the clipboard viewer chain by processing the **ChangeClipboardChain** function before returning from the WM_DESTROY message.

WM_DESTROYCLIPBOARD

This message is sent to the clipboard owner when the clipboard is emptied through a call to the **EmptyClipboard** function.

Parameters *wParam* Is not used.

lParam Is not used.

WM_DEVMODECHANGE

This message is sent to all top-level windows when the user changes device-mode settings.

Parameters *wParam* Is not used.

lParam Points to the device name specified in the Windows initialization file, WIN.INI.

WM_DRAWCLIPBOARD

This message is sent to the first window in the clipboard-viewer chain when the contents of the clipboard change. Only applications that have joined the clipboard-viewer chain by calling the **SetClipboardViewer** function need to process this message.

Parameters *wParam* Is not used.

lParam Is not used.

Comments Each window that receives the WM_DRAWCLIPBOARD message should call the **SendMessage** function to pass the message on to the next window in the clipboard-viewer chain. The handle of the next window is returned by the **SetClipboardViewer** function; it may be modified in response to a WM_CHANGECBCHAIN message.

WM_DRAWITEM 3.0

This message informs the owner-draw button, combo box, list box, or menu that a visual aspect of the control has changed. The **itemAction** field in the **DRAWITEMSTRUCT** structure defines the drawing operation that is to be performed. The data in this field allows the control owner to determine what drawing action is required.

Parameters

wParam Is not used.

lParam Contains a long pointer to a **DRAWITEMSTRUCT** data structure that contains information about the item to be drawn and the type of drawing required.

Comments Before returning from processing this message, an application should restore all objects selected for the display context supplied in the **hDC** field of the **DRAWITEMSTRUCT** data structure.

WM_ENABLE

This message is sent after a window has been enabled or disabled.

Parameters

wParam Specifies whether the window has been enabled or disabled. The *wParam* parameter is nonzero if the window has been enabled; it is zero if the window has been disabled.

lParam Is not used.

WM_ENDSESSION

This message is sent to tell an application that has responded nonzero to a WM_QUERYENDSESSION message whether the session is actually being ended.

Parameters

wParam Specifies whether or not the session is being ended. It is nonzero if the session is being ended. Otherwise, it is zero.

lParam Is not used.

Comments If the *wParam* parameter is nonzero, Windows can terminate any time after all applications have returned from processing this message. Consequently, an application should perform all tasks required for termination before returning from this message.

The application does not need to call the **DestroyWindow** or **PostQuitMessage** function when the session is being ended.

WM_ENTERIDLE

This message informs an application's main windows procedure that a modal dialog box or a menu is entering an idle state. A modal dialog box or menu enters an idle state when no messages are waiting in its queue after it has processed one or more previous messages.

Parameters

wParam Specifies whether the message is the result of a dialog box or a menu being displayed. It is one of these values:

Value	Meaning
MSGF_DIALOGBOX	The system is idle because a dialog box is being displayed.
MSGF_MENU	The system is idle because a menu is being displayed.

lParam Contains in its low-order word the handle of the dialog box (if *wParam* is MSGF_DIALOGBOX) or of the window containing the displayed menu (if *wParam* is MSGF_MENU). The high-order word is not used.

Default action The **DefWindowProc** function returns zero.

WM_ERASEBKGND

This message is sent when the window background needs erasing (for example, when a window is resized). It is sent to prepare an invalidated region for painting.

Parameters

wParam Contains the device-context handle.

lParam Is not used.

Return value The return value is nonzero if the background is erased. Otherwise, it is zero. If the application processes the WM_ERASEBKGND message, it should return the appropriate value.

Default action The background is erased, using the class background brush specified by the **hbrbackground** field in the class structure.

Comments If **hbrbackground** is NULL, the application should process the WM_ERASEBKGND message and erase the background color. When

processing the WM_ERASEBKGND message, the application must align the origin of the intended brush with the window coordinates by first calling the **UnrealizeObject** function for the brush, and then selecting the brush.

Windows assumes the background should be computed by using the MM_TEXT mapping mode. If the device context is using any other mapping mode, the area erased may not be within the visible part of the client area.

WM_FONTCHANGE

This message occurs when the pool of font resources changes. Any application that adds or removes fonts from the system (for example, through the **AddFontResource** or **RemoveFontResource** function) should send this message to all top-level windows.

Parameters *wParam* Is not used.

lParam Is not used.

Comments To send the WM_FONTCHANGE message to all top-level windows, an application can call the **SendMessage** function with the *hWnd* parameter set to 0xFFFF.

WM_GETDLGCODE

This message is sent by Windows to an input procedure associated with a control. Normally, Windows handles all DIRECTION-key and TAB-key input to the control. By responding to the WM_GETDLGCODE message, an application can take control of a particular type of input and process the input itself.

Parameters *wParam* Is not used.

lParam Is not used.

Return value The return value is one or more of the following values, indicating which type of input the application processes:

Value	Meaning
DLGC_DEFPUSHBUTTON	Default push button.
DLGC_HASSETSEL	EM_SETSEL messages.
DLGC_PUSHBUTTON	Push button.
DLGC_RADIOBUTTON	Radio button.

DLGC_WANTALLKEYS	All keyboard input.
DLGC_WANTARROWS	DIRECTION keys.
DLGC_WANTCHARS	WM_CHAR messages.
DLGC_WANTMESSAGE	All keyboard input (the application passes this message on to control).
DLGC_WANTTAB	TAB key.

Default action The **DefWindowProc** function returns zero.

Comments Although the **DefWindowProc** function always returns zero in response to the WM_GETDLGCODE message, the window functions for the predefined control classes return a code appropriate for each class.

The WM_GETDLGCODE message and the returned values are useful only with user-defined dialog controls or standard controls modified by subclassing.

WM_GETFONT 3.0

This message retrieves from a control the font with which the control is currently drawing its text.

Parameters

wParam Not used.

lParam Not used.

Return value The return value is the handle of the font used by the control, or NULL if it is using the system font.

WM_GETMINMAXINFO

This message is sent to a window whenever Windows needs to know the maximized size of the window, the minimum or maximum tracking size of the window, or the maximized position of the window. The maximized size of a window is the size of a window when its borders are fully extended. The maximum tracking size of a window is the largest window size that can be achieved by using the borders to size the window. The minimum tracking size of a window is the smallest window size that can be achieved by using the borders to size the window.

Parameters

wParam Is not used.

lParam Points to an array of five points that contains the following information:

Point	Description
rgpt[0]	Used internally by Windows.
rgpt[1]	The maximized size, which is the screen size by default. The width is (SM_CXSCREEN + (2 × SM_CXFRAME)). The height is (SM_CYSCREEN + (2 × SM_CYFRAME)).
rgpt[2]	The maximized position of the upper-left corner of the window (in screen coordinates). The default *x* value is SM_CXFRAME. The default *y* value is SM_CYFRAME.
rgpt[3]	The minimum tracking size, which is the iconic size by default. The width is SM_CXMINTRACK. The height is SM_CYMINTRACK.
rgpt[4]	The maximum tracking size, which is less than the screen size by default. The width is (SM_CXSCREEN + (2 × SM_CXFRAME)). The height is (SM_CYSCREEN + (2 × SM_CYFRAME)).

Comments The array contains default values for each point before Windows sends the WM_GETMINMAXINFO message. This message gives the application the opportunity to alter the default values.

WM_GETTEXT

This message is used to copy the text that corresponds to a window. For edit controls and combo-box edit controls, the text to be copied is the content of the edit control. For button controls, the text is the button name. For lixt boxes, the text is the currently selected item. For other windows, the text is the window caption.

Parameters

wParam Specifies the maximum number of bytes to be copied, including the null-terminating character.

lParam Points to the buffer that is to receive the text.

Return value The return value is the number of bytes copied. It is LB_ERR if no item is selected or CB_ERR if the combo box has no edit control.

WM_GETTEXTLENGTH

This message is used to find the length (in bytes) of the text associated with a window. The length does not include the null-terminating character. For edit controls and combo-box edit controls, the text is the

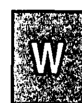
content of the control. For lixt boxes, the text is the currently selected item. For button controls, the text is the button name. For other windows, the text is the window caption.

Parameters *wParam* Is not used.

lParam Is not used.

Comments The return value is the length of the given text.

WM_HSCROLL



This message is sent when the user clicks the horizontal scroll bar.

Parameters *wParam* Contains a scroll-bar code that specifies the user's scrolling request. It can be any one of the following values:

Value	Meaning
SB_BOTTOM	Scroll to lower right.
SB_ENDSCROLL	End scroll.
SB_LINEDOWN	Scroll one line down.
SB_LINEUP	Scroll one line up.
SB_PAGEDOWN	Scroll one page down.
SB_PAGEUP	Scroll one page up.
SB_THUMBPOSITION	Scroll to absolute position. The current position is provided in the low-order word of *lParam*.
SB_THUMBTRACK	Drag thumb to specified position. The current position is provided in the low-order word of *lParam*.
SB_TOP	Scroll to upper left.

lParam Specifies the window handle of the control. If the message is sent by a scroll-bar control, the high-order word of the *lParam* parameter contains the window handle of the control. If the message is sent as a result of the user clicking a pop-up window's scroll bar, the high-order word is not used.

Comments The SB_THUMBTRACK message typically is used by applications that give some feedback while the thumb is being dragged.

If an application scrolls the document in the window, it must also reset the position of the thumb by using the **SetScrollPos** function.

WM_HSCROLLCLIPBOARD

This message is sent when the clipboard contains a data handle for the CF_OWNERDISPLAY format (specifically the clipboard owner should display the clipboard contents) and an event occurs in the clipboard application's horizontal scroll bar.

Parameters *wParam* Contains a handle to the clipboard application window.

lParam Contains one of the following scroll-bar codes in the low-order word:

Value	Meaning
SB_BOTTOM	Scroll to lower right.
SB_ENDSCROLL	End scroll.
SB_LINEDOWN	Scroll one line down.
SB_LINEUP	Scroll one line up.
SB_PAGEDOWN	Scroll one page down.
SB_PAGEUP	Scroll one page up.
SB_THUMBPOSITION	Scroll to absolute position.
SB_TOP	Scroll to upper left.

The high-order word of the *lParam* parameter contains the thumb position if the scroll-bar code is SB_THUMBPOSITION. Otherwise, the high-order word is not used.

Comments The clipboard owner should use the **InvalidateRect** function or repaint as desired. The scroll-bar position should also be reset.

WM_ICONERASEBKGND 3.0

This message is sent to a minimized (iconic) window when the background of the icon must be filled before painting the icon. A window receives this message only if a class icon is defined for the window. Otherwise, WM_ERASEBKGND is sent instead. Passing this message to the **DefWindowProc** function permits Windows to fill the icon background with the background brush of the parent window.

Parameters *wParam* Contains the device-context handle of the icon.

lParam Is not used.

WM_INITDIALOG

This message is sent immediately before a dialog box is displayed. By processing this message, an application can perform any initialization before the dialog box is made visible.

Parameters

wParam Identifies the first control item in the dialog box that can be given the input focus. Generally, this is the first item in the dialog box with WS_TABSTOP style.

lParam Is the value passed as the *dwInitParam* parameter of the function if the dialog box was created by any of the following functions:

- **CreateDialogIndirectParam**
- **CreateDialogParam**
- **DialogBoxIndirectParam**
- **DialogBoxParam**

Otherwise, *lParam* is not used.

Comments If the application returns a nonzero value in response to the WM_INITDIALOG message, Windows sets the input focus to the item identified by the handle in the *wParam* parameter. The application can return FALSE only if it has set the input focus to one of the controls of the dialog box.

WM_INITMENU

This message is a request to initialize a menu. It occurs when a user moves the mouse into a menu bar and clicks, or presses a menu key. Windows sends this message before displaying the menu. This allows the application to change the state of menu items before the menu is shown.

Parameters

wParam Contains the menu handle of the menu that is to be initialized.

lParam Is not used.

Comments A WM_INITMENU message is sent only when a menu is first accessed; only one WM_INITMENU message is generated for each access. This means, for example, that moving the mouse across several menu items while holding down the button does not generate new messages. This message does not provide information about menu items.

WM_INITMENUPOPUP

This message is sent immediately before a pop-up menu is displayed. Processing this message allows an application to change the state of items on the pop-up menu before the menu is shown, without changing the state of the entire menu.

Parameters

wParam Contains the menu handle of the pop-up menu.

lParam Specifies the index of the pop-up menu. The low-order word contains the index of the pop-up menu in the main menu. The high-order word is nonzero if the pop-up menu is the system menu. Otherwise, it is zero.

WM_KEYDOWN

This message is sent when a nonsystem key is pressed. A nonsystem key is a keyboard key that is pressed when the ALT key is *not* pressed, or a keyboard key that is pressed when a window has the input focus.

Parameters

wParam Specifies the virtual-key code of the given key.

lParam Contains the repeat count, scan code, key-transition code, previous key state, and context code, as shown in the following list:

Bit	Value
0–15 (low-order word)	Repeat count (the number of times the key stroke is repeated as a result of the user holding down the key).
16–23 (low byte of high-order word)	Scan code (OEM-dependent value).
24	Extended key, such as a function key or a key on the numeric key pad (1 if it is an extended key).
25–26	Not used.
27–28	Used internally by Windows.
29	Context code (1 if the ALT key is held down while the key is pressed, 0 otherwise).
30	Previous key state (1 if the key is down before the message is sent, 0 if the key is up).

31	Transition state (1 if the key is being released, 0 if the key is being pressed).

For WM_KEYDOWN messages, the key-transition bit (bit 31) is 0 and the context-code bit (bit 29) is 0.

Comments Because of auto-repeat, more than one WM_KEYDOWN message may occur before a WM_KEYUP message is sent. The previous key state (bit 30) can be used to determine whether the WM_KEYDOWN message indicates the first down transition or a repeated down transition.

For IBM Enhanced 101- and 102-key keyboards, enhanced keys are the right ALT and the right CONTROL keys on the main section of the keyboard; the INSERT, DELETE, HOME, END, PAGE UP, PAGE DOWN and DIRECTION keys in the clusters to the left of the numeric key pad; and the divide (/) and ENTER keys in the numeric key pad. Some other keyboards may support the extended-key bit in the *lParam* parameter.

WM_KEYUP

This message is sent when a nonsystem key is released. A nonsystem key is a keyboard key that is pressed when the ALT key is *not* pressed, or a keyboard key that is pressed when a window has the input focus.

Parameters

wParam Specifies the virtual-key code of the given key.

lParam Contains the repeat count, scan code, key-transition code, previous key state, and context code, as shown in the following list:

Bit	Value
0–15 (low-order word)	Repeat count (the number of times the key stroke is repeated as a result of the user holding down the key).
16–23 (low byte of high-order word)	Scan code (OEM-dependent value).
24	Extended key, such as a function key or a key on the numeric key pad (1 if it is an extended key).
25–26	Not used.
27–28	Used internally by Windows.
29	Context code (1 if the ALT key is held down while the key is pressed, 0 otherwise).

30	Previous key state (1 if the key is down before the message is sent, 0 if the key is up).
31	Transition state (1 if the key is being released, 0 if the key is being pressed).

For WM_KEYUP messages, the key-transition bit (bit 31) is 1 and the context-code bit (bit 29) is 0.

Comments For IBM Enhanced 101- and 102-key keyboards, enhanced keys are the right ALT and the right CONTROL keys on the main section of the keyboard; the INSERT, DELETE, HOME, END, PAGE UP, PAGE DOWN and DIRECTION keys in the clusters to the left of the numeric key pad; and the divide (/) and ENTER keys in the numeric key pad. Some other keyboards may support the extended-key bit in the *lParam* parameter.

WM_KILLFOCUS

This message is sent immediately before a window loses the input focus.

Parameters

wParam Contains the handle of the window that receives the input focus (may be NULL).

lParam Is not used.

Comments If an application is displaying a caret, the caret should be destroyed at this point.

WM_LBUTTONDBLCLK

This message occurs when the user double-clicks the left mouse button.

Parameters

wParam Contains a value that indicates whether various virtual keys are down. It can be any combination of the following values:

Value	Meaning
MK_CONTROL	Set if CONTROL key is down.
MK_LBUTTON	Set if left button is down.
MK_MBUTTON	Set if middle button is down.
MK_RBUTTON	Set if right button is down.
MK_SHIFT	Set if SHIFT key is down.

lParam Contains the *x*- and *y*-coordinates of the cursor. The *x*-coordinate is in the low-order word; the *y*-coordinate is in the

high-order word. These coordinates are always relative to the upper-left corner of the window.

Comments Only windows whose window class has CS_DBLCLKS style can receive double-click messages. Windows generates a double-click message when the user presses, releases, and then presses a mouse button again within the system's double-click time limit. Double-clicking actually generates four messages: a down-click message, an up-click message, the double-click message, and another up-click message.

WM_LBUTTONDOWN

This message occurs when the user presses the left mouse button.

Parameters *wParam* Contains a value that indicates whether various virtual keys are down. It can be any combination of the following values:

Value	Meaning
MK_CONTROL	Set if CONTROL key is down.
MK_MBUTTON	Set if middle button is down.
MK_RBUTTON	Set if right button is down.
MK_SHIFT	Set if SHIFT key is down.

lParam Contains the *x*- and *y*-coordinates of the cursor. The *x*-coordinate is in the low-order word; the *y*-coordinate is in the high-order word. These coordinates are always relative to the upper-left corner of the window.

WM_LBUTTONUP

This message occurs when the user releases the left mouse button.

Parameters *wParam* Contains a value that indicates whether various virtual keys are down. It can be any combination of the following values:

Value	Meaning
MK_CONTROL	Set if CONTROL key is down.
MK_MBUTTON	Set if middle button is down.
MK_RBUTTON	Set if right button is down.
MK_SHIFT	Set if SHIFT key is down.

lParam Contains the *x*- and *y*-coordinates of the cursor. The *x*-coordinate is in the low-order word; the *y*-coordinate is in the

high-order word. These coordinates are always relative to the upper-left corner of the window.

WM_MBUTTONDBLCLK

This message occurs when the user double-clicks the middle mouse button.

Parameters

wParam Contains a value that indicates whether various virtual keys are down. It can be any combination of the following values:

Value	Meaning
MK_CONTROL	Set if CONTROL key is down.
MK_LBUTTON	Set if left button is down.
MK_MBUTTON	Set if middle button is down.
MK_RBUTTON	Set if right button is down.
MK_SHIFT	Set if SHIFT key is down.

lParam Contains the *x*- and *y*-coordinates of the cursor. The *x*-coordinate is in the low-order word; the *y*-coordinate is in the high-order word. These coordinates are always relative to the upper-left corner of the window.

Comments

Only windows whose window class has CS_DBLCLKS style can receive double-click messages. Windows generates a double-click message when the user presses, releases, and then presses a mouse button again within the system's double-click time limit. Double-clicking actually generates four messages: a down-click message, an up-click message, the double-click message, and another up-click message.

WM_MBUTTONDOWN

This message occurs when the user presses the middle mouse button.

Parameters

wParam Contains a value that indicates whether various virtual keys are down. It can be any combination of the following values:

Value	Meaning
MK_CONTROL	Set if CONTROL key is down.
MK_LBUTTON	Set if left button is down.
MK_RBUTTON	Set if right button is down.
MK_SHIFT	Set if SHIFT key is down.

lParam Contains the *x*- and *y*-coordinates of the cursor. The *x*-coordinate is in the low-order word; the *y*-coordinate is in the high-order word. These coordinates are always relative to the upper-left corner of the window.

WM_MBUTTONUP

This message occurs when the user releases the middle mouse button.

Parameters *wParam* Contains a value that indicates whether various virtual keys are down. It can be any combination of the following values:

Value	Meaning
MK_CONTROL	Set if CONTROL key is down.
MK_LBUTTON	Set if left button is down.
MK_RBUTTON	Set if right button is down.
MK_SHIFT	Set if SHIFT key is down.

lParam Contains the *x*- and *y*-coordinates of the cursor. The *x*-coordinate is in the low-order word; the *y*-coordinate is in the high-order word. These coordinates are always relative to the upper-left corner of the window.

WM_MDIACTIVATE 3.0

An application sends this message to a multiple document interface (MDI) client window to instruct the client window to activate a different MDI child window. As the client window processes this message, it sends WM_MDIACTIVATE to the child window being deactivated and to the child window being activated.

Parameters *wParam* When the application sends the WM_MDIACTIVATE message to its MDI client window, the *wParam* parameter contains the window handle of the MDI child window to be activated. When the client window sends the message to a child window, *wParam* is TRUE if the child is being activated and FALSE if it is being deactivated.

lParam When received by an MDI child window, the *lParam* parameter contains in its high-order word the window handle of the child window being deactivated and in its low-order word the window handle of the child window being activated. When

this message is sent to the client window, *lParam* should be set to NULL.

Comments MDI child windows are activated independently of the MDI frame window. When the frame becomes active, the child window that was last activated with the WM_MDIACTIVE message receives the WM_NCACTIVATE message to draw an active window frame and caption bar, but it does not receive another WM_MDIACTIVATE message.

WM_MDICASCADE 3.0

This message arranges the child windows of a multiple document interface (MDI) client window in a "cascade" format.

Parameters

wParam	Not used.
lParam	Not used.

WM_MDICREATE 3.0

This message causes a multiple document interface (MDI) client window to create a child window.

Parameters

wParam	Not used.
lParam	Contains a long pointer to an **MDICREATESTRUCT** data structure.

Return value The return value contains the identifier of the new window in the low word and zero in the high word.

Comments The window is created with the style bits WS_CHILD, WS_CLIPSIBLINGS, WS_CLIPCHILDREN, WS_SYSMENU, WS_CAPTION, WS_THICKFRAME, WS_MINIMIZEBOX, and WS_MAXIMIZEBOX, plus additional style bits specified in the **MDICREATESTRUCT** data structure to which *lParam* points. Windows adds the title of the new child window to the window menu of the frame window. An application should create all child windows of the client window with this message.

If a client window receives any message that changes the activation of child windows and the currently active MDI child window is maximized, Windows restores the currently active child and maximizes the newly activated child.

When the MDI child window is created, Windows sends the WM_CREATE message to the window. The *lParam* parameter of the WM_CREATE message contains a pointer to a **CREATESTRUCT** data structure. The **lpCreateParams** field of the **CREATESTRUCT** structure contains a pointer to the **MDICREATESTRUCT** data structure passed with the WM_MDICREATE message that created the MDI child window.

An application should not send a second WM_MDICREATE message while a WM_MDICREATE message is still being processed. For example, it should not send a WM_MDICREATE message while an MDI child window is processing its WM_CREATE message.

WM_MDIDESTROY 3.0

When sent to a multiple document interface (MDI) client window, this message causes a child window to be closed.

Parameters *wParam* Contains the window handle of the child window.

lParam Not used.

Comments This message removes the title of the child window from the frame window and deactivates the child window. An application should close all MDI child windows with this message.

If a client window receives any message that changes the activation of child windows and the currently active MDI child window is maximized, Windows restores the currently active child and maximizes the newly activated child.

WM_MDIGETACTIVE 3.0

This message returns the current active multiple document interface (MDI) child window, along with a flag indicating whether the child is maximized or not.

Parameters *wParam* Not used.

lParam Not used.

Return value The return value contains the handle of the active MDI child window in its low word. If the window is maximized, the high word contains 1; otherwise, the high word is zero.

WM_MDIICONARRANGE 3.0

This message is sent to a multiple document interface (MDI) client window to arrange all minimized document child windows. It does not affect child windows that are not minimized.

Parameters

wParam Not used.

lParam Not used.

WM_MDIMAXIMIZE 3.0

This message causes a multiple document interface (MDI) client window to maximize an MDI child window. When a child window is maximized, Windows resizes it to make its client area fill the client window. Windows places the child window's System menu in the frame's menu bar so that the user can restore or close the child window and adds the title of the child window to the frame window title.

Parameters

wParam Contains the window identifier of the child window.

lParam Not used.

Comments

If an MDI client window receives any message that changes the activation of its child windows, and if the currently active MDI child window is maximized, Windows restores the currently active child and maximizes the newly activated child.

WM_MDINEXT 3.0

This message activates the next multiple document interface (MDI) child window immediately behind the currently active child window and places the currently active window behind all other child windows.

Parameters

wParam Not used.

lParam Not used.

Comments

If an MDI client window receives any message that changes the activation of its child windows, and if the currently active MDI child window is maximized, Windows restores the currently active child and maximizes the newly activated child.

WM_MDIRESTORE 3.0

This message restores a multiple document interface (MDI) child window from maximized or minimized size.

Parameters *wParam* Contains the window identifier of the child window.

lParam Not used.

WM_MDISETMENU 3.0

This message replaces the menu of a multiple document interface (MDI) frame window, the Window pop-up menu, or both.

Parameters *wParam* Not used.

lParam Contains in its low-order word the menu handle (**HMENU**) of the new frame-window menu, and contains in its high-order word the menu handle of the new Window pop-up menu. If either word is zero, the corresponding menu is not changed.

Return value The return value is the handle of the frame-window menu replaced by this message.

Comments After sending this message, an application must call the **DrawMenuBar** function to update the menu bar.

If this message replaces the Window pop-up menu, MDI child-window menu items are removed from the previous Window menu and added to the new Window pop-up menu.

If an MDI child window is maximized and this message replaces the MDI frame-window menu, the System menu and restore controls are removed from the previous frame-window menu and added to the new menu.

WM_MDITILE 3.0

This message causes a multiple document interface (MDI) client window to arrange all its child windows in a tiled format.

Parameters *wParam* Not used.

lParam Not used.

WM_MEASUREITEM 3.0

This message is sent to the owner of an owner-draw button, combo box, list box, or menu item when the control is created. When the owner receives the message, the owner fills in the **MEASUREITEM** data structure pointed to by the *lParam* message parameter and returns; this informs Windows of the dimensions of the control.

If a list box or combo box is created with the LBS_OWNERDRAWVARIABLE or CBS_OWNERDRAWVARIABLE style, this message is sent to the owner for each item in the control. Otherwise, this message is sent once.

Parameters

wParam Not used.

lParam Contains a long pointer to a **MEASUREITEMSTRUCT** data structure that contains the dimensions of the owner-draw control.

Comments Windows sends the WM_MEASUREITEM message to the owner of combo boxes and list boxes created with the OWNERDRAWFIXED style before sending WM_INITDIALOG.

WM_MENUCHAR

This message is sent when the user presses a menu mnemonic character that doesn't match any of the predefined mnemonics in the current menu. It is sent to the window that owns the menu.

Parameters

wParam Contains the ASCII character that the user pressed.

lParam The high-order word contains a handle to the selected menu. The low-order word contains the MF_POPUP flag if the menu is a pop-up menu. It contains the MF_SYSMENU flag if the menu is a System menu.

Return value The high-order word of the return value contains one of the following command codes:

Code	Meaning
0	Informs Windows that it should discard the character that the user pressed, and creates a short beep on the system speaker.
1	Informs Windows that it should close the current menu.
2	Informs Windows that the low-order word of the return value contains the menu item-number for a specific item. This item is selected by Windows.

The low-order word is ignored if the high-order word contains zero or 1. Applications should process this message when accelerators are used to select bitmaps placed in a menu.

WM_MENUSELECT

This message occurs when the user selects a menu item.

Parameters

wParam Identifies the item selected. If the selected item is a menu item, *wParam* contains the menu-item ID. If the selected item contains a pop-up menu, *wParam* contains the handle of the pop-up menu.

lParam The low-order word contains a combination of the following menu flags:

Value	Meaning
MF_BITMAP	Item is a bitmap.
MF_CHECKED	Item is checked.
MF_DISABLED	Item is disabled.
MF_GRAYED	Item is grayed.
MF_MOUSESELECT	Item was selected with a mouse.
MF_OWNERDRAW	Item is an owner-draw item.
MF_POPUP	Item contains a pop-up menu.
MF_SYSMENU	Item is contained in the System menu. The high-order word identifies the menu associated with the message.

Comments If the low-order word of the *lParam* parameter contains –1 and the high-order word contains zero, Windows has closed the menu because the user pressed ESC or clicked outside the menu. In this case, *wParam* will also contain zero.

WM_MOUSEACTIVATE

This message occurs when the cursor is in an inactive window and any mouse button is pressed. The parent receives this message only if the child passes it to the **DefWindowProc** function.

Parameters

wParam Contains a handle to the topmost parent window of the window being activated.

lParam Contains the hit-test area code in the low-order word and the mouse message number in the high-order word. A hit test is a test that determines the location of the cursor.

Return value The return value specifies whether the window should be activated and whether the mouse event should be discarded. It must be one of the following values:

Value	Meaning
MA_ACTIVATE	Activate the window.
MA_NOACTIVATE	Do not activate the window.
MA_ACTIVATEANDEAT	Activate the window and discard the mouse event.

Comments If the child window passes the message to the **DefWindowProc** function, **DefWindowProc** passes this message to a window's parent window before any processing occurs. If the parent window returns TRUE, processing is halted.

For a description of the individual hit-test area codes, see Table 6.2, "Hit-test codes."

WM_MOUSEMOVE

This message occurs when the user moves the mouse.

Parameters *wParam* Contains a value that indicates whether various virtual keys are down. It can be any combination of the following values:

Value	Meaning
MK_CONTROL	Set if CONTROL key is down.
MK_LBUTTON	Set if left button is down.
MK_MBUTTON	Set if middle button is down.
MK_RBUTTON	Set if right button is down.
MK_SHIFT	Set if SHIFT key is down.

lParam Contains the *x*- and *y*-coordinates of the cursor. The *x*-coordinate is in the low-order word; the *y*-coordinate is in the high-order word. These coordinates are always relative to the upper-left corner of the window.

Comments The **MAKEPOINT** macro can be used to convert the *lParam* parameter to a **POINT** structure.

WM_MOVE

This message is sent after a window has been moved.

Parameters *wParam* Is not used.

lParam Contains the new location of the upper-left corner of the client area of the window. This new location is given in screen coordinates for overlapped and pop-up windows and parent-client coordinates for child windows. The *x*-coordinate is in the low-order word; the *y*-coordinate is in the high-order word.

WM_NCACTIVATE

This message is sent to a window when its nonclient area needs to be changed to indicate an active or inactive state.

Parameters *wParam* Specifies when a caption bar or icon needs to be changed to indicate an active or inactive state. The *wParam* parameter is nonzero if an active caption or icon is to be drawn. It is zero for an inactive caption or icon.

lParam Is not used.

Default action The **DefWindowProc** function draws a gray caption bar for an inactive window and a black caption bar for an active window.

WM_NCCALCSIZE

This message is sent when the size of a window's client area needs to be calculated.

Parameters *wParam* Is not used.

lParam Points to a **RECT** data structure that contains the screen coordinates of the window rectangle (including client area, borders, caption, scroll bars, and so on).

Default action The **DefWindowProc** function calculates the size of the client area, based on the window characteristics (presence of scroll bars, menu, and so on), and places the result in the **RECT** data structure.

WM_NCCREATE

This message is sent prior to the WM_CREATE message when a window is first created.

Parameters *wParam* Contains a handle to the window that is being created.

lParam Points to the **CREATESTRUCT** data structure for the window.

Return value The return value is nonzero if the nonclient area is created. It is zero if an error occurs; the **CreateWindow** function will return NULL in this case.

Default action Scroll bars are initialized (the scroll-bar position and range are set) and the window text is set. Memory used internally to create and maintain the window is allocated.

WM_NCDESTROY

This message informs a window that its nonclient area is being destroyed. The **DestroyWindow** function sends the WM_NCDESTROY message to the window following the WM_DESTROY message. This message is used to free the allocated memory block associated with the window.

Parameters *wParam* Is not used.

lParam Is not used.

Default action This message frees any memory internally allocated for the window.

WM_NCHITTEST

This message is sent to the window that contains the cursor (or the window that used the **GetCapture** function to capture the mouse input) every time the mouse is moved.

Parameters *wParam* Is not used.

lParam Contains the *x*- and *y*-coordinates of the cursor. The *x*-coordinate is in the low-order word; the *y*-coordinate is in the high-order word. These coordinates are always screen coordinates.

Return value The return value of the **DefWindowProc** function is one of the values given in Table 6.2, indicating the position of the cursor:

Table 6.2
Hit-test codes

Code	Meaning
HTBOTTOM	In the lower horizontal border of window.
HTBOTTOMLEFT	In the lower-left corner of window border.
HTBOTTOMRIGHT	In the lower-right corner of window border.
HTCAPTION	In a caption area.
HTCLIENT	In a client area.
HTERROR	Same as HTNOWHERE except that the **DefWindowProc** function produces a system beep to indicate an error.
HTGROWBOX	In a size box.
HTHSCROLL	In the horizontal scroll bar.
HTLEFT	In the left border of window.
HTMENU	In a menu area.
HTNOWHERE	On the screen background or on a dividing line between windows.
HTREDUCE	In a minimize box.
HTRIGHT	In the right border of window.
HTSIZE	Same as HTGROWBOX.
HTSYSMENU	In a control-menu box (close box in child windows).
HTTOP	In the upper horizontal border of window.
HTTOPLEFT	In the upper-left corner of window border.
HTTOPRIGHT	In the upper-right corner of window border.
HTTRANSPARENT	In a window currently covered by another window.
HTVSCROLL	In the vertical scroll bar.
HTZOOM	In a maximize box.

Comments The **MAKEPOINT** macro can be used to convert the *lParam* parameter to a **POINT** structure.

WM_NCLBUTTONDBLCLK

This message is sent to a window when the user double-clicks the left mouse button while the cursor is within a nonclient area of the window.

Parameters

wParam Contains the code returned by WM_NCHITTEST (for more information, see the WM_NCHITTEST message, earlier in this chapter).

lParam Contains a **POINT** data structure that contains the *x*- and *y*-screen coordinates of the cursor position. These coordinates are always relative to the upper-left corner of the screen.

Default action If appropriate, WM_SYSCOMMAND messages are sent.

WM_NCLBUTTONDOWN

This message is sent to a window when the user presses the left mouse button while the cursor is within a nonclient area of the window.

Parameters

wParam Contains the code returned by WM_NCHITTEST (for more information, see the WM_NCHITTEST message, earlier in this chapter).

lParam Contains a **POINT** data structure that contains the *x*- and *y*-screen coordinates of the cursor position. These coordinates are always relative to the upper-left corner of the screen.

Default action If appropriate, WM_SYSCOMMAND messages are sent.

WM_NCLBUTTONUP

This message is sent to a window when the user releases the left mouse button while the cursor is within a nonclient area of the window.

Parameters

wParam Contains the code returned by WM_NCHITTEST (for more information, see the WM_NCHITTEST message, earlier in this chapter).

lParam Contains a **POINT** data structure that contains the *x*- and *y*-screen coordinates of the cursor position. These coordinates are always relative to the upper-left corner of the screen.

Default action If appropriate, WM_SYSCOMMAND messages are sent.

WM_NCMBUTTONDBLCLK

This message is sent to a window when the user double-clicks the middle mouse button while the cursor is within a nonclient area of the window.

Parameters

wParam Contains the code returned by WM_NCHITTEST (for more information, see the WM_NCHITTEST message, earlier in this chapter).

lParam Contains a **POINT** data structure that contains the *x*- and *y*-screen coordinates of the cursor position. These coordinates are always relative to the upper-left corner of the screen.

WM_NCMBUTTONDOWN

This message is sent to a window when the user presses the middle mouse button while the cursor is within a nonclient area of the window.

Parameters

wParam Contains the code returned by WM_NCHITTEST (for more information, see the WM_NCHITTEST message, earlier in this chapter).

lParam Contains a **POINT** data structure that contains the *x*- and *y*-screen coordinates of the cursor position. These coordinates are always relative to the upper-left corner of the screen.

WM_NCMBUTTONUP

This message is sent to a window when the user releases the left mouse button while the cursor is within a nonclient area of the window.

Parameters

wParam Contains the code returned by WM_NCHITTEST (for more information, see the WM_NCHITTEST message, earlier in this chapter).

lParam Contains a **POINT** data structure that contains the *x*- and *y*-screen coordinates of the cursor position. These coordinates are always relative to the upper-left corner of the screen.

WM_NCMOUSEMOVE

This message is sent to a window when the cursor is moved within a nonclient area of the window.

Parameters

wParam Contains the code returned by WM_NCHITTEST (for more information, see the WM_NCHITTEST message, earlier in this chapter).

lParam Contains a **POINT** data structure that contains the *x*- and *y*-screen coordinates of the cursor position. These coordinates are always relative to the upper-left corner of the screen.

Default action If appropriate, WM_SYSCOMMAND messages are sent.

WM_NCPAINT

This message is sent to a window when its frame needs painting.

Parameters *wParam* Is not used.

lParam Is not used.

Default action The **DefWindowProc** function paints the window frame.

Comments An application can intercept this message and paint its own custom window frame. Remember that the clipping region for a window is always rectangular, even if the shape of the frame is altered.

WM_NCRBUTTONDBLCLK

This message is sent to a window when the user double-clicks the right mouse button while the cursor is within a nonclient area of the window.

Parameters *wParam* Contains the code returned by WM_NCHITTEST (for more information, see the WM_NCHITTEST message, earlier in this chapter).

lParam Contains a **POINT** data structure that contains the *x*- and *y*-screen coordinates of the cursor position. These coordinates are always relative to the upper-left corner of the screen.

WM_NCRBUTTONDOWN

This message is sent to a window when the user presses the right mouse button while the cursor is within a nonclient area of the window.

Parameters *wParam* Contains the code returned by WM_NCHITTEST (for more information, see the WM_NCHITTEST message, earlier in this chapter).

lParam Contains a **POINT** data structure that contains the *x*- and *y*-screen coordinates of the cursor position. These coordinates are always relative to the upper-left corner of the screen.

WM_NCRBUTTONUP

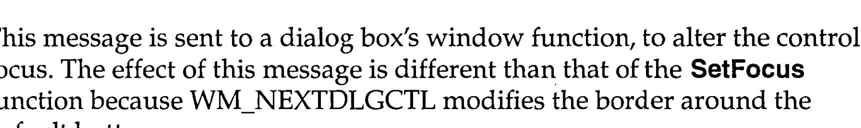
This message is sent to a window when the user releases the right mouse button while the cursor is within a nonclient area of the window.

Parameters

wParam Contains the code returned by WM_NCHITTEST (for more information, see the WM_NCHITTEST message, earlier in this chapter).

lParam Contains a **POINT** data structure that contains the *x*- and *y*-screen coordinates of the cursor position. These coordinates are always relative to the upper-left corner of the screen.

WM_NEXTDLGCTL



This message is sent to a dialog box's window function, to alter the control focus. The effect of this message is different than that of the **SetFocus** function because WM_NEXTDLGCTL modifies the border around the default button.

Parameters

wParam If the *lParam* parameter is nonzero, the *wParam* parameter identifies the control that receives the focus. If *lParam* is zero, *wParam* is a flag that indicates whether the next or previous control with tab-stop style receives the focus. If *wParam* is zero, the next control receives the focus; otherwise, the previous control with tab-stop style receives the focus.

lParam Contains a flag that indicates how Windows uses the *wParam* parameter. If the *lParam* parameter is nonzero, *wParam* is a handle associated with the control that receives the focus; otherwise, *wParam* is a flag that indicates whether the next or previous control with tab-stop style receives the focus.

Comments Do not use the **SendMessage** function to send a WM_NEXTDLGCTL message if your application will concurrently process other messages that set the control focus. Use the **PostMessage** function instead.

WM_PAINT

This message is sent when Windows or an application makes a request to repaint a portion of an application's window. The message is sent either when the **UpdateWindow** function is called or by the **DispatchMessage**

function when the application obtains a WM_PAINT message by using the **GetMessage** or **PeekMessage** function.

Parameters *wParam* Is not used.

lParam Is not used.

WM_PAINTCLIPBOARD

This message is sent when the clipboard contains a data handle for the CF_OWNERDISPLAY format (specifically the clipboard owner should display the clipboard contents) and the Clipboard application's client area needs repainting. The WM_PAINTCLIPBOARD message is sent to the clipboard owner to request repainting of all or part of the Clipboard application's client area.

Parameters *wParam* Contains a handle to the Clipboard-application window.

lParam The low-order word of the *lParam* parameter identifies a **PAINTSTRUCT** data structure that defines what part of the client area to paint. The high-order word is not used.

Comments To determine whether the entire client area or just a portion of it needs repainting, the clipboard owner must compare the dimensions of the drawing area given in the **rcpaint** field of the **PAINTSTRUCT** data structure to the dimensions given in the most recent WM_SIZECLIPBOARD message.

An application must use the **GlobalLock** function to lock the memory that contains the **PAINTSTRUCT** data structure. The application should unlock that memory by using the **GlobalUnlock** function before it yields or returns control.

WM_PAINTICON 3.0

This message is sent to a minimized (iconic) window when the icon is to be painted. A window receives this message only if a class icon is defined for the window. Otherwise, WM_PAINT is sent instead. Passing this message to the **DefWindowProc** function permits Windows to paint the icon with the class icon.

Parameters *wParam* Is not used.

lParam Is not used.

WM_PALETTECHANGED 3.0

This message informs all windows that the window with input focus has realized its logical palette, thereby changing the system palette. This message allows windows without input focus that use a color palette to realize their logical palettes and update their client areas.

Parameters *wParam* Contains the handle of the window that caused the system palette to change.

lParam Is not used.

Comments To avoid creating a loop, a window that receives this message should not realize its palette unless it determines that *wParam* does not contain its window handle.

WM_PARENTNOTIFY 3.0

This message is sent to the parent of a child window when the child window is created or destroyed, or when the user has pressed a mouse button while the cursor is over the child window. When the child window is being created, Windows sends WM_PARENTNOTIFY just before the **CreateWindow** or **CreateWindowEx** function that creates the window returns. When the child window is being destroyed, Windows sends the message before any processing to destroy the window takes place.

Parameters *wParam*, Specifies the event for which the parent is being notified. It can be any of these values:

Value	**Meaning**
WM_CREATE	The child window is being created.
WM_DESTROY	The child window is being destroyed.
WM_LBUTTONDOWN WM_MBUTTONDOWN WM_RBUTTONDOWN	The user has clicked on a child window.

lParam Contains the window handle of the child window in its low-order word and the ID of the child window in its high-order word if *wParam* is WM_CREATE or WM_DESTROY. Otherwise, *lParam* contains the *x*- and *y*-coordinates of the cursor. The *x*-coordinate is in the low-order word and the *y*-coordinate is in the high-order word.

Comments This message is also sent to all ancestor windows of the child window, including the top-level window.

This message is sent to the parent of all child windows unless the child has the extended window style WS_EX_NOPARENTNOTIFY; **CreateWindowEx** creates a window with extended window styles. By default, child windows in a dialog box have the WS_EX_NOPARENTNOTIFY style unless the child window was created by calling the **CreateWindowEx** function.

WM_PASTE

This message inserts the data from the clipboard into the control window at the current cursor position. Data are inserted only if the clipboard contains data in CF_TEXT format.

Parameters

wParam Is not used.

lParam Is not used.

WM_QUERYDRAGICON 3.0

This message is sent to a minimized (iconic) window which is about to be dragged by the user but which does not have an icon defined for its class.

When the user drags the icon of a window without a class icon, Windows replaces the icon with a default icon cursor. If the application needs a different cursor to be displayed during dragging, it must return the handle of a monochrome cursor compatible with the display driver's resolution. The application can call the **LoadCursor** function to load a cursor from the resources in its executable file and to obtain this handle.

Parameters

wParam Is not used.

lParam Is not used.

Return value The return value contains in its low-order word the handle of the cursor which Windows is to display while the user drags the icon. The return value is NULL if Windows is to display the default icon cursor. The default return value is NULL.

WM_QUERYENDSESSION

This message is sent when the user chooses the End Session command. If any application returns zero, the session is not ended. Windows stops sending WM_QUERYENDSESSION messages as soon as one application returns zero, and sends WM_ENDSESSION messages, with the *wParam* parameter set to zero, to any applications that have already returned nonzero.

Parameters *wParam* Is not used.

lParam Is not used.

Return value The return value is nonzero if the application can be conveniently shut down. Otherwise, it is zero.

Default action The **DefWindowProc** function returns nonzero.

WM_QUERYNEWPALETTE 3.0

This message informs a window that it is about to receive input focus. If the window realizes its logical palette when it receives input focus, the window should return TRUE; otherwise, it should return FALSE.

Parameters *wParam* Is not used.

lParam Is not used.

Return value The return value is TRUE if the window realizes its logical palette. Otherwise, it is FALSE.

WM_QUERYOPEN

This message is sent to an icon when the user requests that it be opened into a window.

Parameters *wParam* Is not used.

lParam Is not used.

Return value The return value is zero when the application prevents the icon from being opened. It is nonzero when the icon can be opened.

Default action The **DefWindowProc** function returns nonzero.

WM_QUIT

This message indicates a request to terminate an application and is generated when the application calls the **PostQuitMessage** function. It causes the **GetMessage** function to return zero.

Parameters *wParam* Contains the exit code given in the **PostQuitMessage** call.

lParam Is not used.

WM_RBUTTONDBLCLK

This message occurs when the user double-clicks the right mouse button.

Parameters *wParam* Contains a value that indicates whether various virtual keys are down. It can be any combination of the following values:

Value	Meaning
MK_CONTROL	Set if CONTROL key is down.
MK_LBUTTON	Set if left button is down.
MK_MBUTTON	Set if middle button is down.
MK_RBUTTON	Set if right button is down.
MK_SHIFT	Set if SHIFT key is down.

lParam Contains the *x*- and *y*-coordinates of the cursor. The *x*-coordinate is in the low-order word; the *y*-coordinate is in the high-order word. These coordinates are always relative to the upper-left corner of the window.

Comments Only windows whose window class has CS_DBLCLKS style can receive double-click messages. Windows generates a double-click message when the user presses, releases, and then presses a mouse button again within the system's double-click time limit. Double-clicking actually generates four messages: a down-click message, an up-click message, the double-click message, and another up-click message.

WM_RBUTTONDOWN

This message occurs when the user presses the right mouse button.

Parameters *wParam* Contains a value that indicates whether various virtual keys are down. It can be any combination of the following values:

Value	Meaning
MK_CONTROL	Set if CONTROL key is down.
MK_LBUTTON	Set if left button is down.
MK_MBUTTON	Set if middle button is down.
MK_SHIFT	Set if SHIFT key is down.

lParam Contains the *x*- and *y*-coordinates of the cursor. The *x*-coordinate is in the low-order word; the *y*-coordinate is in the high-order word. These coordinates are always relative to the upper-left corner of the window.

WM_RBUTTONUP

This message occurs when the user releases the right mouse button.

Parameters *wParam* Contains a value that indicates whether various virtual keys are down. It can be any combination of the following values:

Value	Meaning
MK_CONTROL	Set if CONTROL key is down.
MK_LBUTTON	Set if left button is down.
MK_MBUTTON	Set if middle button is down.
MK_SHIFT	Set if SHIFT key is down.

lParam Contains the *x*- and *y*-coordinates of the cursor. The *x*-coordinate is in the low-order word; the *y*-coordinate is in the high-order word. These coordinates are always relative to the upper-left corner of the window.

WM_RENDERALLFORMATS

This message is sent to the application that owns the clipboard when that application is being destroyed.

Parameters *wParam* Is not used.

lParam Is not used.

Comments The application should render the clipboard data in all the formats it is capable of generating and pass a handle to each format to the **SetClipboardData** function. This ensures that the data in the clipboard can be rendered even though the application has been destroyed.

WM_RENDERFORMAT

This message requests that the clipboard owner format the data last copied to the clipboard in the specified format, and then pass a handle to the formatted data to the clipboard.

Parameters *wParam* Specifies the data format. It can be any one of the formats described with the **SetClipboardData** function.

lParam Is not used.

WM_SETCURSOR

This message occurs if mouse input is not captured and the mouse causes cursor movement within a window.

Parameters *wParam* Contains a handle to the window that contains the cursor.

lParam Contains the hit-test area code in the low-order word and the mouse message number in the high-order word.

Comments The **DefWindowProc** function passes the WM_SETCURSOR message to a parent window before processing. If the parent window returns TRUE, further processing is halted. Passing the message to a window's parent window gives the parent window control over the cursor's setting in a child window. The **DefWindowProc** function also uses this message to set the cursor to an arrow if it is not in the client area, or to the registered-class cursor if it is. If the low-order word of the *lParam* parameter is HTERROR and the high-order word of *lParam* is a mouse button-down message, the **MessageBeep** function is called.

The high-order word of *lParam* is zero when the window enters menu mode.

WM_SETFOCUS

This message is sent after a window gains the input focus.

Parameters *wParam* Contains the handle of the window that loses the input focus (may be NULL).

lParam Is not used.

Comments To display a caret, an application should call the appropriate caret functions at this point.

WM_SETFONT 3.0

This message specifies the font that a dialog box control is to use when drawing text. The best time for the owner of a dialog box control to set the font of the control is when it receives the WM_INITDIALOG message. The application should call the **DeleteObject** function to delete the font when it is no longer needed, such as after the control is destroyed.

The size of the control is not changed as a result of receiving this message. To prevent Windows from clipping text that does not fit within the boundaries of the control, the application should correct the size of the control window before changing the font.

Parameters

wParam Contains the handle of the font. If this parameter is NULL, the control will draw text using the default system font.

lParam Specifies whether the control should be redrawn immediately upon setting the font. Setting *lparam* to TRUE causes the control to redraw itself; otherwise, it will not.

Comments Before Windows creates a dialog box with the DS_SETFONT style, Windows sends the WM_SETFONT message to the dialog-box window before creating the controls. An application creates a dialog box with the DS_SETFONT style by calling any of the following functions:

- **CreateDialogIndirect**
- **CreateDialogIndirectParam**
- **DialogBoxIndirect**
- **DialogBoxIndirectParam**

The **DLGTEMPLATE** data structure which the application passes to these functions must have the DS_SETFONT style set and must contain a **FONTINFO** data structure that defines the font with which the dialog box will draw text.

WM_SETREDRAW

This message is sent by an application to a window in order to allow changes in that window to be redrawn, or to prevent changes in that window from being redrawn.

Parameters

wParam Specifies the state of the redraw flag. If the *wParam* parameter is nonzero, the redraw flag is set. If *wParam* is zero, the flag is cleared.

lParam Is not used.

Comments This message sets or clears the redraw flag. However, it does not direct a list box to update its display. When the redraw flag is set, a control can be redrawn immediately after each change. When the redraw flag is cleared, no redrawing is done. Applications that need to add several names to a list without redrawing until the final name is added should set the redraw flag before adding the final name to the list.

WM_SETTEXT

This message is used to set the text of a window. For edit controls and combo-box edit controls, the text to be set is the content of the edit control. For button controls, the text is the button name. For other windows, the text is the window caption.

Parameters

wParam Is not used.

lParam Points to a null-terminated string that is the window text.

Return value The return value is LB_ERRSPACE (for a list box) or CB_ERRSPACE (for a combo box) if insufficient space is available to set the text in the edit control. It is CB_ERR if this message is sent to a combo box without an edit control.

Comments This message does not change the current selection in the list box of a combo box. An application should use the CB_SELECTSTRING message to select the list-box item which matches the text in the edit control.

WM_SHOWWINDOW

This message is sent when a window is to be hidden or shown. A window is hidden or shown when the **ShowWindow** function is called; when an overlapped window is maximized or restored; or when an overlapped or pop-up window is closed (made iconic) or opened (displayed on the screen). When an overlapped window is closed, all pop-up windows associated with that window are hidden.

Parameters

wParam Specifies whether a window is being shown. It is nonzero if the window is being shown. It is zero if the window is being hidden.

lParam Specifies the status of the window being shown. It is zero if the message is sent because of a **ShowWindow** function call. Otherwise, the *lParam* parameter is one of the following values:

Value	Meaning
SW_PARENTCLOSING	Parent window is closing (being made iconic) or a pop-up window is being hidden.
SW_PARENTOPENING	Parent window is opening (being displayed) or a pop-up window is being shown.

Default action The **DefWindowProc** function hides or shows the window as specified by the message.

WM_SIZE

This message is sent after the size of a window has changed.

Parameters *wParam* Contains a value that defines the type of resizing requested. It can be one of the following values:

Value	Meaning
SIZEFULLSCREEN	Window has been maximized.
SIZEICONIC	Window has been minimized.
SIZENORMAL	Window has been resized, but neither SIZEICONIC nor SIZEFULLSCREEN applies.
SIZEZOOMHIDE	Message is sent to all pop-up windows when some other window is maximized.
SIZEZOOMSHOW	Message is sent to all pop-up windows when some other window has been restored to its former size.

lParam Contains the new width and height of the client area of the window. The width is in the low-order word; the height is in the high-order word.

Comments If the **SetScrollPos** or **MoveWindow** function is called for a child window as a result of the WM_SIZE message, the *bRedraw* parameter should be nonzero to cause the window to be repainted.

WM_SIZECLIPBOARD

This message is sent when the clipboard contains a data handle for the CF_OWNERDISPLAY format (that is, the clipboard owner should display

the clipboard contents) and the clipboard-application window has changed size.

Parameters

wParam Identifies the clipboard-application window.

lParam The low-order word of the *lParam* parameter identifies a **RECT** data structure that specifies the area the clipboard owner should paint. The high-order word is not used.

Comments A WM_SIZECLIPBOARD message is sent with a null rectangle (0,0,0,0) as the new size when the clipboard application is about to be destroyed or minimized. This permits the clipboard owner to free its display resources.

An application must use the **GlobalLock** function to lock the memory that contains the **PAINTSTRUCT** data structure. The application should unlock that memory by using the **GlobalUnlock** function before it yields or returns control.

WM_SPOOLERSTATUS 3.0

This message is sent from Print Manager whenever a job is added to or removed from the Print Manager queue.

Parameters

wParam Is set to SP_JOBSTATUS.

lparam Specifies in its low-order word the number of jobs remaining in the Print Manager queue. The high-order word is not used.

Comments This message is for informational purposes only.

WM_SYSCHAR

This message results when a WM_SYSKEYUP and WM_SYSKEYDOWN message are translated. It specifies the virtual-key code of the System-menu key.

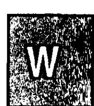
Parameters

wParam Contains the ASCII-character key code of a System-menu key.

lParam Contains the repeat count, scan code, key-transition code, previous key state, and context code, as shown in the following list:

Bit	Value
0–15 (low-order word)	Repeat count (the number of times the key stroke is repeated as a result of the user holding down the key).

16–23 (low byte of high-order word)	Scan code (OEM-dependent value).
24	Extended key, such as a function key or a key on the numeric key pad (1 if it is an extended key, 0 otherwise).
25–26	Not used.
27–28	Used internally by Windows.
29	Context code (1 if the ALT key is held down while the key is pressed, 0 otherwise).
30	Previous key state (1 if the key is down before the message is sent, 0 if the key is up).
31	Transition state (1 if the key is being released, 0 if the key is being pressed).



Default action None.

Comments When the context code is zero, the message can be passed to the **TranslateAccelerator** function, which will handle it as though it were a normal key message instead of a system-key message. This allows accelerator keys to be used with the active window even if the active window does not have the input focus.

For IBM Enhanced 101- and 102-key keyboards, enhanced keys are the right ALT and the right CONTROL keys on the main section of the keyboard; the INSERT, DELETE, HOME, END, PAGE UP, PAGE DOWN and DIRECTION keys in the clusters to the left of the numeric key pad; and the divide (/) and ENTER keys in the numeric key pad. Some other keyboards may support the extended-key bit in the *lParam* parameter.

WM_SYSCOLORCHANGE

This message specifies a change in one or more system colors. Windows sends the message to all top-level windows when a change is made in the system color setting.

Parameters

wParam Is not used.

lParam Is not used.

Default action Windows sends a WM_PAINT message to any window that is affected by a system color change.

Comments Applications that have brushes that use the existing system colors should delete those brushes and re-create them by using the new system colors.

WM_SYSCOMMAND

This message is sent when the user selects a command from the System menu or when the user selects the maximize or minimize box.

Parameters

wParam Specifies the type of system command requested. It can be any one of the following values:

Value	Meaning
SC_CLOSE	Close the window.
SC_HOTKEY	Activate a window in response to the user pressing a hotkey.
SC_HSCROLL	Scroll horizontally.
SC_KEYMENU	Retrieve a menu through a key stroke.
SC_MAXIMIZE (or SC_ZOOM)	Maximize the window.
SC_MINIMIZE (or SC_ICON)	Minimize the window.
SC_MOUSEMENU	Retrieve a menu through a mouse click.
SC_MOVE	Move the window.
SC_NEXTWINDOW	Move to the next window.
SC_PREVWINDOW	Move to the previous window.
SC_RESTORE	Checkpoint (save the previous coordinates).
SC_SCREENSAVE	Executes or activates the Windows Screen Saver application.
SC_SIZE	Size the window.
SC_TASKLIST	Executes or activates the Windows Task Manager application.
SC_VSCROLL	Scroll vertically.

lParam Contains the cursor coordinates if a System-menu command is chosen with the mouse. The low-order word contains the *x*-coordinate, and the high-order word contains the *y*-coordinate. If *wParam* is SC_HOTKEY, the low-order word contains the handle of the window to be activated. Otherwise, this parameter is not used.

Default action The **DefWindowProc** function carries out the System-menu request for the predefined actions specified above.

Comments In WM_SYSCOMMAND messages, the four low-order bits of the *wParam* parameter are used internally by Windows. When an application tests the value of *wParam*, it must combine the value 0xFFF0 with the *wParam* value by using the bitwise AND operator to obtain the correct result.

The menu items in a System menu can be modified by using the **GetSystemMenu**, **AppendMenu**, **InsertMenu**, and **ModifyMenu** functions. Applications that modify the System menu must process WM_SYSCOMMAND messages. Any WM_SYSCOMMAND messages not handled by the application must be passed to the **DefWindowProc** function. Any command values added by an application must be processed by the application and cannot be passed to **DefWindowProc**.

An application can carry out any system command at any time by passing a WM_SYSCOMMAND message to the **DefWindowProc** function.

Accelerator key strokes that are defined to select items from the System menu are translated into WM_SYSCOMMAND messages; all other accelerator key strokes are translated into WM_COMMAND messages.

WM_SYSDEADCHAR

This message results when a WM_SYSKEYUP and WM_SYSKEYDOWN message are translated. It specifies the character value of a dead key.

Parameters

wParam	Contains the dead-key character value.
lParam	Contains a repeat count and an auto-repeat count. The low-order word contains the repeat count; the high-order word contains the auto-repeat count.

WM_SYSKEYDOWN

This message is sent when the user holds down the ALT key and then presses another key. It also occurs when no window currently has the input focus; in this case, the WM_SYSKEYDOWN message is sent to the active window. The window that receives the message can distinguish between these two contexts by checking the context code in the *lParam* parameter.

Parameters

wParam	Contains the virtual-key code of the key being pressed.

lParam Contains the repeat count, scan code, key-transition code, previous key state, and context code, as shown in the following list:

Bit	Value
0–15 (low-order word)	Repeat count (the number of times the key stroke is repeated as a result of the user holding down the key).
16–23 (low byte of high-order word)	Scan code (OEM-dependent value).
24	Extended key, such as a function key or a key on the numeric key pad (1 if it is an extended key).
25–26	Not used.
27–28	Used internally by Windows.
29	Context code (1 if the ALT key is held down while the key is pressed, 0 otherwise).
30	Previous key state (1 if the key is down before the message is sent, 0 if the key is up).
31	Transition state (1 if the key is being released, 0 if the key is being pressed).

For WM_SYSKEYDOWN messages, the key-transition bit (bit 31) is 0. The context-code bit (bit 29) is 1 if the ALT key is down while the key is pressed; it is 0 if the message is sent to the active window because no window has the input focus.

Comments When the context code is zero, the message can be passed to the **TranslateAccelerator** function, which will handle it as though it were a normal key message instead of a system-key message. This allows accelerator keys to be used with the active window even if the active window does not have the input focus.

Because of auto-repeat, more than one WM_SYSKEYDOWN message may occur before a WM_SYSKEYUP message is sent. The previous key state (bit 30) can be used to determine whether the WM_SYSKEYDOWN message indicates the first down transition or a repeated down transition.

For IBM Enhanced 101- and 102-key keyboards, enhanced keys are the right ALT and the right CONTROL keys on the main section of the keyboard; the INSERT, DELETE, HOME, END, PAGE UP, PAGE DOWN and DIRECTION keys in the clusters to the left of the numeric key pad; and the divide (/) and

ENTER keys in the numeric key pad. Some other keyboards may support the extended-key bit in the *lParam* parameter.

WM_SYSKEYUP

This message is sent when the user releases a key that was pressed while the ALT key was held down. It also occurs when no window currently has the input focus; in this case, the WM_SYSKEYUP message is sent to the active window. The window that receives the message can distinguish between these two contexts by checking the context code in the *lParam* parameter.

Parameters *wParam* Contains the virtual-key code of the key being released.

lParam Contains the repeat count, scan code, key-transition code, previous key state, and context code, as shown in the following list:

Bit	Value
0–15 (low-order word)	Repeat count (the number of times the key stroke is repeated as a result of the user holding down the key).
16–23 (low byte of high-order word)	Scan code (OEM-dependent value).
24	Extended key, such as a function key or a key on the numeric key pad (1 if it is an extended key).
25–26	Not used.
27–28	Used internally by Windows.
29	Context code (1 if the ALT key is held down while the key is pressed, 0 otherwise).
30	Previous key state (1 if the key is down before the message is sent, 0 if the key is up).
31	Transition state (1 if the key is being released, 0 if the key is being pressed).

For WM_SYSKEYUP messages, the key-transition bit (bit 31) is 1. The context-code bit (bit 29) is 1 if the ALT key is down while the key is pressed; it is 0 if the message is sent to the active window because no window has the input focus.

Comments When the context code is zero, the message can be passed to the **TranslateAccelerator** function, which will handle it as though it were a normal key message instead of a system-key message. This allows accelerator keys to be used with the active window even if the active window does not have the input focus.

For IBM Enhanced 101- and 102-key keyboards, enhanced keys are the right ALT and the right CONTROL keys on the main section of the keyboard; the INSERT, DELETE, HOME, END, PAGE UP, PAGE DOWN and DIRECTION keys in the clusters to the left of the numeric key pad; and the divide (/) and ENTER keys in the numeric key pad. Some other keyboards may support the extended-key bit in the *lParam* parameter.

For non-USA Enhanced 102-key keyboards, the right ALT key is handled as a CONTROL-ALT key. The following shows the sequence of messages which result when the user presses and releases this key:

Order	Message	Virtual-key code (*lParam*)
1	WM_KEYDOWN	VK_CONTROL
2	WM_KEYDOWN	VK_MENU
3	WM_KEYUP	VK_CONTROL
4	WM_SYSKEYUP	VK_MENU

WM_TIMECHANGE

This message occurs when an application makes a change (or set of changes) to the system time. Any application that changes the system time should send this message to all top-level windows.

Parameters *wParam* Is not used.

lParam Is not used.

Comments To send the WM_TIMECHANGE message to all top-level windows, an application can use the **SendMessage** function with the *hWnd* parameter set to 0xFFFF.

WM_TIMER

This message occurs when the time limit set for a given timer has elapsed.

Parameters *wParam* Contains the timer ID, an integer value that identifies the timer.

lParam Points to a function that was passed to the **SetTimer** function when the timer was created. If the *lParam* parameter is not NULL, Windows calls the specified function directly, instead of sending the WM_TIMER message to the window function.

WM_UNDO

This message undoes the last operation. When sent to an edit control, the previously deleted text is restored or the previously added text is deleted.

Parameters

wParam Is not used.

lParam Is not used.

WM_VKEYTOITEM

This message is sent by a list box with the LBS_WANTKEYBOARDINPUT style to its owner in response to a WM_KEYDOWN message.

Parameters

wParam Contains the virtual-key code of the key which the user pressed.

lParam Contains the current caret position in its high-order word and the window handle of the list box in its low-order word.

Return value

The return value specifies the action which the application performed in response to the message. A return value of –2 indicates that the application handled all aspects of selecting the item and wants no further action by the list box. A return value of –1 indicates that the list box should perform the default action in response to the key stroke. A return value of zero or greater specifies the index of an item in the list box and indicates that the list box should perform the default action for the key stroke on the given item.

WM_VSCROLL

This message is sent when the user clicks the vertical scroll bar.

Parameters

wParam Contains a scroll-bar code that specifies the user's scrolling request. It can be any one of the following values:

Value	Meaning
SB_BOTTOM	Scroll to bottom.
SB_ENDSCROLL	End scroll.
SB_LINEDOWN	Scroll one line down.
SB_LINEUP	Scroll one line up.
SB_PAGEDOWN	Scroll one page down.
SB_PAGEUP	Scroll one page up.
SB_THUMBPOSITION	Scroll to absolute position. The current position is provided in the low-order word of *lParam*.
SB_THUMBTRACK	Drag thumb to specified position. The current position is provided in the low-order word of *lParam*.
SB_TOP	Scroll to top.

lParam If the message is sent by a scroll-bar control, the high-order word of the *lParam* parameter identifies the control. If the message is sent as a result of the user clicking a pop-up window's scroll bar, the high-order word is not used.

Comments The SB_THUMBTRACK message typically is used by applications that give some feedback while the thumb is being dragged.

If an application scrolls the document in the window, it must also reset the position of the thumb by using the **SetScrollPos** function.

WM_VSCROLLCLIPBOARD

This message is sent when the clipboard contains a data handle for the CF_OWNERDISPLAY format (that is, the clipboard owner should display the clipboard contents) and an event occurs in the clipboard-application's vertical scroll bar.

Parameters *wParam* Contains a handle to the clipboard-application window.

lParam Contains one of the following scroll-bar codes in the low-order word:

Value	Meaning
SB_BOTTOM	Scroll to lower right.
SB_ENDSCROLL	End scroll.
SB_LINEDOWN	Scroll one line down.
SB_LINEUP	Scroll one line up.
SB_PAGEDOWN	Scroll one page down.
SB_PAGEUP	Scroll one page up.

SB_THUMBPOSITION	Scroll to absolute position.
SB_TOP	Scroll to upper left.

The high-order word of the *lParam* parameter contains the thumb position if the scroll-bar code is SB_THUMBPOSITION. Otherwise, the high-order word is not used.

Comments The clipboard owner should use the **InvalidateRect** function or repaint as desired. The scroll bar position should also be reset.

WM_WININICHANGE

This message is sent when the Windows initialization file, WIN.INI, changes. Any application that makes a change to WIN.INI should send this message to all top-level windows.

Parameters

wParam Is not used.

lParam Points to a string that specifies the name of the section that has changed (the string does not include the square brackets).

Comments To send the WM_WININICHANGE message to all top-level windows, an application can use the **SendMessage** function with the *hWnd* parameter set to 0xFFFF.

Although it is incorrect to do so, some applications send this message with *lParam* set to NULL. If an application receives this message with a NULL *lParam*, it should check all sections in WIN.INI that affect the application.

I N D E X

[[ ]] (double brackets)
[#double brackets] as document convention *9*
{ } (curly braces)
[#curly braces]as document convention *9*
( ) (parentheses)
[#parentheses]as document convention *8*
... (ellipses)
[#ellipses]as document convention *9*
| (vertical bar)
[#vertical bar] as document convention *9*
\bc169\ec \bc170\ec (quotation marks)
[#quotation marks]as document convention[(quotation marks), as document convention] *9*
\bcB\ecBold text\bcD\ec
[#bold text]as document convention *8*
\bcF105M\ecMonospaced type\bcF255D\ec
[#monospaced type]as document convention *9*
\bcMI\ecItalic text\bcD\ec
[#italic text]as document convention *9*
_FPInit function *282*
_FPTerm function *283*
_lclose function *401*
_lclose function[#lclose function] *144*
_lcreat function *401*
_lcreat function[#lcreat function] *144*
_llseek function *404*
_llseek function[#llseek function] *144*
_lopen function *422*
_lopen function[#lopen function] *144*
_lread function *424*
_lread function[#lread function] *144*
_lwrite function *427*
_lwrite function[#lwrite function] *144*

A

AccessResource function *138, 147*
AddAtom function *140, 148*
AddFontResource function *113, 148*
AdjustWindowRect function *19, 149*
AdjustWindowRectEx function *19, 150*
Aligning brushes *53*
AllocDStoCSAlias function *136, 150*
AllocResource function *138, 151*
AllocSelector function *136, 152*
ALTERNATE filling mode *202, 510*
ALTERNATE polygon-filling mode *202, 334, 509*
AnimatePalette function *95, 152*
ANSI_FIXED_FONT stock object *343*
ANSI_VAR_FONT stock object *343*
AnsiLower function *139, 153*
AnsiLowerBuff function *139, 153*
AnsiNext function *139, 154*
AnsiPrev function *139, 154*
AnsiToOem function *139, 154*
AnsiToOemBuff function *139, 155*
AnsiUpper function *139, 155*
AnsiUpperBuff function *139, 156*
AnyPopup function *73, 156*
AppendMenu function *39, 72, 157, 198, 203, 347, 691*
application does not process. *20*
Arc function *107, 159*
ArrangeIconicWindows function *42, 160*
ASPECTX device capability *305*
ASPECTXY device capability *306*
ASPECTY device capability *305*

B

Background:brush@class *25*
BeginDeferWindowPos function *42, 160*
BeginPaint function *44, 161*
BitBlt function *110, 162*
Bitmap functions *110, 111*

BITMAPINFO data structure *189, 309, 497*
BITMAPINFOHEADER data structure *189*
BITSPIXEL device capability *305*
BLACK_BRUSH stock object *343*
BLACK_PEN stock object *343*
BLACKNESS raster-operation code *163*
BLACKONWHITE stretching mode *518*
BM_GETCHECK message *593, 603*
BM_GETSTATE message *593, 603*
BM_SETCHECK message *593, 603*
BM_SETSTATE message *593, 604*
BM_SETSTYLE message *593, 604*
BN_CLICKED message *597, 605*
BN_DOUBLECLICKED message *597, 606*
Braces
  curly ({ })
    as document convention *9*
Brackets
  double ([[ ]])
    as document convention *9*
BringWindowToTop function *42, 164*
Brush
  origin
    default *46*
BS_3STATE control style *212, 605*
BS_AUTO3STATE control style *211, 604*
BS_AUTOCHECKBOX control style *211, 604*
BS_AUTORADIOBUTTON control style *604*
BS_CHECKBOX control style *211, 604*
BS_DEFPUSHBUTTON control style *211, 605*
BS_GROUPBOX control style *211, 605*
BS_LEFTTEXT control style *211, 605*
BS_OWNERDRAW control style *212, 605, 606*
BS_PUSHBUTTON control style *212, 605*
BS_RADIOBUTTON control style *212, 605, 606*
BuildCommDCB function *142, 164*
Button
  owner-draw *65*
BUTTON control class *207, 211*
Button notification codes *597*

C

Cache
  display-context *50*
CallMsgFilter function *79, 165*
CallWindowProc function *14, 28, 166*
Capital letters
  small
    as document convention *9*
Catch function *138, 166*
CB_ADDSTRING message *595, 606, 607, 608, 609, 611*
CB_DELETESTRING message *595, 606, 648*
CB_DIR message *595, 607*
CB_FINDSTRING message *607*
CB_FINDSTRING message *596*
CB_GETCOUNT message *596, 608*
CB_GETCURSEL message *596, 608*
CB_GETEDITSEL message *596, 608*
CB_GETITEMDATA message *596, 609*
CB_GETLBTEXT message *596, 609*
CB_GETLBTEXTLEN message *596, 609*
CB_INSERTSTRING message *596, 607, 608, 609, 610, 611*
CB_LIMITTEXT message *596, 610*
CB_RESETCONTENT message *596, 610, 648*
CB_SELECTSTRING message *596, 611*
CB_SETCURSEL message *596, 611*
CB_SETEDITSEL message *596, 612*
CB_SETITEMDATA message *596, 609, 612*
CB_SHOWDROPDOWN message *596, 612*
CBN_DBLCLK message *598, 613*
CBN_DROPDOWN message *598, 613*
CBN_EDITCHANGE message *598, 613*
CBN_EDITUPDATE message *598, 614*
CBN_ERRSPACE message *598, 614*
CBN_KILLFOCUS message *598, 614*
CBN_SELCHANGE message *598, 615*
CBN_SETFOCUS message *598, 615*
CBS_HASSTRINGS control style *212, 606, 607, 608, 609, 610, 611*
CBS_OEMCONVERT control style *212*
CBS_OWNERDRAWFIXED control style *213*
CBS_OWNERDRAWVARIABLE control style *213*
CE_BREAK communication error code *300*
CE_CTSTO communication error code *300*
CE_DNS communication error code *300*
CE_DSRTO communication error code *300*
CE_FRAME communication error code *300*
CE_IOE communication error code *300*
CE_MODE communication error code *300*
CE_OOP communication error code *300*

CE_OVERRUN communication error code *300*
CE_PTO communication error code *300*
CE_RLSDTO communication error code *301*
CE_RXOVER communication error code *301*
CE_RXPARITY communication error code *301*
CE_TXFULL communication error code *301*
CF_BITMAP clipboard format *492*
CF_DIB clipboard format *492*
CF_DIF clipboard format *492*
CF_DSPBITMAP clipboard format *492*
CF_DSPMETAFILEPICT clipboard format *492*
CF_DSPTEXT clipboard format *492*
CF_METAFILEPICT clipboard format *492*
CF_OEMTEXT clipboard format *492*
CF_OWNERDISPLAY clipboard format *492*
CF_PALETTE clipboard format *296, 492*
CF_SYLK clipboard format *493*
CF_TEXT clipboard format *493*
CF_TIFF clipboard format *493*
ChangeClipboardChain function *74, 167*
ChangeMenu function *167*
ChangeSelector function *136, 168*
Character cell *115*
CheckDlgButton function *57, 168*
CheckMenuItem function *72, 169, 436*
CheckRadioButton function *57, 170*
ChildWindowFromPoint function *73, 105, 171*
Chord function *109, 110, 171*
Class background brush *25*
Class icon *25*
Class menu *26*
ClearCommBreak function *142, 172*
CLIENTCREATESTRUCT data structure *38, 600*
ClientToScreen function *105, 172*
Clipboard
  getting prioritized format *334*
Clipboard formats *491*
CLIPCAPS device capability *306*
ClipCursor function *77, 173*
Clipping functions *107, 108*
Clipping region
  default *46*
CloseClipboard function *74, 173*
CloseComm function *142, 174*
CloseMetaFile function *124, 174*
CloseSound function *142, 174*
CloseWindow function *42, 175*
CLRDTR communication function code *269*
CLRRTS communication function code *269*
COLOR_ACTIVEBORDER system color *519*
COLOR_ACTIVECAPTION system color *26, 519*
COLOR_APPWORKSPACE system color *26, 519*
COLOR_BACKGROUND system color *26, 519*
COLOR_BTNFACE system color *26, 519*
COLOR_BTNSHADOW system color *26, 519*
COLOR_BTNTEXT system color *520*
COLOR_CAPTIONTEXT system color *26, 520*
COLOR_GRAYTEXT system color *26, 383, 520*
COLOR_HIGHLIGHT system color *26, 520*
COLOR_HIGHLIGHTTEXT system color *26, 520*
COLOR_INACTIVEBORDER system color *520*
COLOR_INACTIVECAPTION system color *26, 520*
COLOR_MENU system color *26, 520*
COLOR_MENUTEXT system color *26, 520*
Color palette
  default *46*
COLOR_SCROLLBAR system color *26, 520*
COLOR_WINDOW system color *26, 520*
COLOR_WINDOWFRAME system color *26, 520*
COLOR_WINDOWTEXT system color *26, 520*
COLORONCOLOR stretching mode *518*
COLORREF data type *94, 144*
CombineRgn function *106, 175*
Combo box *65*
COMBOBOX control class *208*
COMPAREITEMSTRUCT data structure *645*
COMPLEXREGION region type *176, 270, 298, 358, 391, 442, 443, 481*
Coordinate functions *105*
CopyMetaFile function *124, 176*
CopyRect function *83, 176*
CountClipboardFormats function *177*
CountVoiceNotes function *142, 177*
CreateBitmap function *110, 177*
CreateBitmapIndirect function *110, 178*
CreateBrushIndirect function *91, 179*
CreateCaret function *75, 179*
CreateCompatibleBitmap function *110, 180*

CreateCompatibleDC function *88, 181*
CreateCursor function *77, 182*
CreateDC function *88, 182*
CreateDialog function *57, 60, 183, 188, 220*
CreateDialogIndirect function *58, 185, 220, 685*
CreateDialogIndirectParam function *58, 187, 220, 685*
CreateDialogParam function *58, 188, 220*
CreateDIBitmap function *111, 189*
CreateDIBPatternBrush function *91, 190*
CreateDiscardableBitmap function *110, 191*
CreateEllipticRgn function *106, 192*
CreateEllipticRgnIndirect function *106, 192*
CreateFont function *113, 193*
CreateFontIndirect function *113, 195*
CreateHatchBrush function *91, 196*
CreateIC function *88, 196*
CreateIcon function *54, 197*
CreateMenu function *72, 198*
CreateMetaFile function *124, 198*
CreatePalette function *95, 153, 199, 483*
CreatePatternBrush function *91, 199*
CreatePen function *91, 200*
CreatePenIndirect function *91, 200*
CreatePolygonRgn function *106, 201*
CreatePolyPolygonRgn function *106, 201*
CreatePopupMenu function *39, 72, 202*
CreateRectRgn function *106, 203*
CreateRectRgnIndirect function *106, 203*
CreateRoundRectRgn function *106, 204*
CreateSolidBrush function *91, 204*
CreateWindow function *15*
CreateWindowEx function *19, 150, 218, 219*
CS_BYTEALIGNCLIENT window class style *27*
CS_BYTEALIGNWINDOW window class style *27*
CS_CLASSDC window class style *27, 47*
CS_DBLCLKS window class style *27*
CS_GLOBALCLASS window class style *27*
CS_HREDRAW window class style *27*
CS_NOCLOSE window class style *27*
CS_OWNDC window class style *27, 29, 48*
CS_PARENTDC window class style *27*
CS_SAVEBITS window class style *27*
CS_VREDRAW window class style *27*
CTLCOLOR_BTN control type for setting color *646*
CTLCOLOR_DLG control type for setting color *646*
CTLCOLOR_EDIT control type for setting color *646*
CTLCOLOR_LISTBOX control type for setting color *646*
CTLCOLOR_MSGBOX control type for setting color *646*
CTLCOLOR_SCROLLBAR control type for setting color *646*
CTLCOLOR_STATIC control type for setting color *646*
Curly braces ({ })
as document convention *9*
CURVECAPS device capability *306*

D

DC_BINS device capability *232*
DC_DRIVER device capability *233*
DC_DUPLEX device capability *233*
DC_EXTRA device capability *233*
DC_FIELDS device capability *233*
DC_MAXEXTENT device capability *233*
DC_MINEXTENT device capability *233*
DC_PAPERS device capability *233*
DC_PAPERSIZE device capability *234*
DC_SIZE device capability *234*
DC_VERSION device capability *234*
DebugBreak function *145, 219*
DEFAULT_PALETTE stock object *344*
DefDlgProc function *19, 58, 220*
DeferWindowPos function *42, 221*
DefFrameProc function *19, 222*
DefHookProc function *79, 224*
DefineHandleTable function *134, 137, 224*
DefMDIChildProc function *19, 225*
DefWindowProc function *226*
DeleteAtom function *140, 227*
DeleteDC function *88, 181, 227*
DeleteMenu function *72, 228*
DeleteMetaFile function *124, 229*
DeleteObject function *91, 229, 685*
DestroyCaret function *75, 230*
DestroyCursor function *77, 230*
DestroyMenu function *72, 231*
DestroyWindow function *20*

Device context *88*
DEVICE_DEFAULT_FONT stock object *343*
DeviceCapabilities function *128, 232*
DeviceMode function *128, 235*
DEVMODE data structure *233, 234, 272*
Dialog functions *57*
DialogBox function *58, 60, 220, 235*
DialogBoxIndirect function *58, 220, 237, 685*
DialogBoxIndirectParam function *58, 220, 239, 685*
DialogBoxParam function *58, 220, 239*
Digitized aspect
  fonts *119*
DispatchMessage function *14, 240*
Display
  updating *51*
Display context default characteristics *46*
DKGRAY_BRUSH stock object *343*
DLGC_DEFPUSHBUTTON input type *652*
DLGC_HASSETSEL input type *652*
DLGC_PUSHBUTTON input type *652*
DLGC_RADIOBUTTON input type *652*
DLGC_WANTALLKEYS input type *653*
DLGC_WANTARROWS input type *653*
DLGC_WANTCHARS input type *653*
DLGC_WANTMESSAGE input type *653*
DLGC_WANTTAB input type *653*
DlgDirList function *58, 241*
DlgDirListComboBox function *58, 242*
DlgDirSelect function *58, 244*
DlgDirSelectComboBox function *58, 244*
DLGTEMPLATE data structure *685*
DM_COPY option *272*
DM_GETDEFID message *593, 615*
DM_MODIFY option *272*
DM_PROMPT option *272*
DM_SETDEFID message *593, 615*
DM_UPDATE option *272*
DOS3Call function *137, 245*
DOS interrupt function request (21H) *245*
Double brackets ([[ ]])
  as document convention *9*
DPtoLP function *105, 246*
DrawFocusRect function *44, 109, 247*
DrawIcon function *44, 247*
DrawMenuBar function *72, 158, 228, 248, 389, 436, 471*
DrawText function *44, 248*
DRIVERVERSION device capability *305*
DS_LOCALEDIT dialog-box style *209*
DS_MODALFRAME dialog-box style *209*
DS_NOIDLEMSG dialog-box style *209*
DS_SETFONT dialog-box style *685*
DS_SYSMODAL dialog-box style *209*
DSTINVERT raster operation *163*
DT_BOTTOM format for DrawText function *250*
DT_CALCRECT format for DrawText function *250*
DT_EXPANDTABS format for DrawText function *250*
DT_EXTERNALLEADING format for DrawText function *250*
DT_NOCLIP format for DrawText function *250*
DT_NOPREFIX format for DrawText function *250*
DT_SINGLELINE format for DrawText function *250*
DT_TABSTOP format for DrawText function *250*
DT_TOP format for DrawText function *250*
DT_VCENTER format for DrawText function *250*
DT_WORDBREAK format for DrawText function *250*

E

EDIT control class *208*
Edit-control notification codes *597*
Ellipse and polygon functions *109*
Ellipse function *109, 110, 251*
EM_CANUNDO message *593, 616*
EM_EMPTYUNDOBUFFER message *593, 616*
EM_FMTLINES message *593, 616*
EM_GETHANDLE message *593, 617*
EM_GETLINE message *593, 617*
EM_GETLINECOUNT message *593, 617*
EM_GETMODIFY message *593, 618*
EM_GETRECT message *593, 618*
EM_GETSEL message *593, 618*
EM_LIMITTEXT message *593, 618*
EM_LINEFROMCHAR message *594, 619*
EM_LINEINDEX message *594, 619*

EM_LINELENGTH message *594, 619*
EM_LINESCROLL message *594, 620*
EM_REPLACESEL message *594, 620*
EM_SETHANDLE message *594, 620*
EM_SETMODIFY message *594, 621*
EM_SETPASSWORDCHAR message *214, 594, 621*
EM_SETRECT message *594, 621*
EM_SETRECTNP message *594, 622*
EM_SETSEL message *594, 622*
EM_SETTABSTOPS message *594, 622*
EM_SETWORDBREAK message *594, 623*
EM_UNDO message *594, 624*
EmptyClipboard function *74, 252*
EMS memory
  determining available *315*
EN_CHANGE message *597, 624*
EN_ERRSPACE message *597, 625*
EN_HSCROLL message *597, 625*
EN_KILLFOCUS message *597, 625*
EN_MAXTEXT message *597, 626*
EN_SETFOCUS message *597, 626*
EN_UPDATE message *597, 626*
EN_VSCROLL message *597, 627*
EnableHardwareInput function *44, 252*
EnableMenuItem function *72, 253, 436*
EnableWindow function *43, 254*
EndDeferWindowPos function *42, 254*
EndDialog function *58, 240, 255*
EndPaint function *44, 255*
EnumChildWindows function *73, 256*
EnumClipboardFormats function *74, 257*
EnumFonts function *113, 258*
EnumMetaFile function *124, 260*
EnumObjects function *91, 261*
ENUMPAPERBINS printer escape *233*
EnumProps function *81, 262*
EnumTaskWindows function *73, 265*
EnumWindows function *73, 266*
EqualRect function *83, 267*
EqualRgn function *106, 267*
ERROR region type *176, 270, 298, 358, 391, 442, 443, 481*
ES_AUTOHSCROLL control style *213*
ES_AUTOVSCROLL control style *213*
ES_CENTER control style *213*
ES_LEFT control style *213*
ES_LOWERCASE control style *213*
ES_MULTILINE control style *213*
ES_NOHIDESEL control style *214*
ES_OEMCONVERT control style *214*
ES_PASSWORD control style *214*
ES_RIGHT control style *214*
ES_UPPERCASE control style *214*
Escape function *268*
EscapeCommFunction function *142, 269*
EV_BREAK event-mask value *494*
EV_CTS event-mask value *494*
EV_DSR event-mask value *494*
EV_ERR event-mask value *494*
EV_PERR event-mask value *494*
EV_RING event-mask value *494*
EV_RLSD event-mask value *495*
EV_RXCHAR event-mask value *495*
EV_RXFLAG event-mask value *495*
EV_TXEMPTY event-mask value *495*
ExcludeClipRect function *107, 269*
ExcludeUpdateRgn function *44, 270*
ExitWindows function *138, 271*
ExtDeviceMode function *128, 271*
ExtFloodFill function *110, 273*
ExtTextOut function *112, 274*

F

FatalAppExit function *145, 276*
FatalExit function *145, 276*
FillRect function *44, 277*
FillRgn function *106, 278*
Filters
  installing *80*
FindAtom function *140, 278*
FindResource function *138, 278*
FindWindow function *73, 280*
Fixed-pitch font attribute *119*
FlashWindow function *75, 280*
FloodFill function *110, 281*
FlushComm function *142, 282*
Font
  family *114*
Font functions *113*
Font Selection *124*
FONTINFO data structure *685*

Formats
  clipboard *491*
Formatted text
  styles *55*
FrameRect function *44, 283*
FrameRgn function *106, 284*
FreeLibrary function *134, 284*
FreeModule function *134, 285*
FreeProcInstance function *134, 285*
FreeResource function *138, 285*
FreeSelector function *137, 286*
Functions
  clipping *107*
  coordinates *105*
  device context attributes *88*
  device contexts *88*
  environment *131*
  font *113*
  metafile *124*
  printer control *128*
  text *112*

G

GCL_MENUNAME option *489*
GCL_WNDPROC option *293, 489*
GCW_CBCLSEXTRA option *294, 490*
GCW_CBWNDEXTRA option *295, 490*
GCW_HBRBACKGROUND option *295, 490*
GCW_HCURSOR option *295, 490*
GCW_HICON option *295, 490*
GCW_HMODULE option *295*
GCW_STYLE option *295, 490*
GetActiveWindow function *43, 286*
GetAspectRatioFilter function *119, 287*
GetAsyncKeyState function *44, 287*
GetAtomHandle function *140, 287*
GetAtomName function *140, 288*
GetBitmapBits function *110, 288*
GetBitmapDimension function *110, 289*
GetBkColor function *99, 289*
GetBkMode function *99, 289*
GetBrushOrg function *91, 290*
GetBValue function *94, 290*
GetCapture function *43, 290*
GetCaretBlinkTime function *75, 291*
GetCaretPos function *75, 291*
GetCharWidth function *113, 291*
GetClassInfo function *20, 292*
GetClassLong function *20, 293*
GetClassName function *20, 294*
GetClassWord function *20, 294*
GetClientRect function *42, 295*
GetClipboardData function *74, 295*
GetClipboardFormatName function *74, 296*
GetClipboardOwner function *74, 297*
GetClipboardViewer function *74, 297*
GetClipBox function *107, 297*
GetCodeHandle function *134, 298*
GetCodeInfo function *137, 298*
GetCommError function *142, 300*
GetCommEventMask function *142, 301*
GetCommState function *142, 301*
GetCurrentPDB function *138, 302*
GetCurrentPosition function *302*
GetCurrentTask function *138, 302*
GetCurrentTime function *43, 74, 303*
GetCursorPos function *77, 303*
GetDC function *45, 303*
GetDCOrg function *88, 304*
GetDesktopWindow function *304*
GetDeviceCaps function *305*
GetDialogBaseUnits function *58, 61, 215, 308, 430, 623, 637*
GetDIBits function *99, 111, 306, 309*
GetDlgCtrlID function *58, 310*
GetDlgItem function *58, 310*
GetDlgItemInt function *58, 311*
GetDlgItemText function *58, 312*
GetDOSEnvironment function *138, 312*
GetDoubleClickTime function *43, 313*
GetDriveType function *144, 313*
GetEnvironment function *131, 313*
GetFocus function *43, 314*
GetFreeSpace function *135, 315*
GetGValue function *94, 316*
GetInputState function *44, 316*
GetInstanceData function *134, 316*
GetKBCodePage function *44, 317*
GetKeyboardState function *44, 317*
GetKeyboardType function *318*
GetKeyNameText function *44, 319*
GetKeyState function *44, 320*
GetLastActivePopup function *20, 320*

GetMapMode function *100, 321*
GetMenu function *72, 321*
GetMenuCheckMarkDimensions function *72, 321*
GetMenuItemCount function *72, 322*
GetMenuItemID function *72, 322*
GetMenuState function *72, 322*
GetMenuString function *72, 323*
GetMessage function *14, 324*
GetMessagePos function *14, 326*
GetMessageTime function *14, 326*
GetMetaFile function *124, 327*
GetMetaFileBits function *124, 327*
GetModuleFileName function *134, 327*
GetModuleHandle function *134, 328*
GetModuleUsage function *134, 328*
GetNearestColor function *94, 329*
GetNearestPaletteIndex function *95, 329*
GetNextDlgGroupItem function *58, 329*
GetNextDlgTabItem function *58, 330*
GetNextWindow function *73, 330*
GetNumTasks function *138, 331*
GetObject function *91, 331*
GetPaletteEntries function *95, 332*
GetParent function *73, 333*
GetPixel function *110, 333*
GetPolyFillMode function *99, 334*
GetPriorityClipboardFormat function *74, 334*
GetPrivateProfileInt function *141, 335*
GetPrivateProfileString function *141, 336, 337*
GetProcAddress function *134, 337*
GetProfileInt function *141, 338*
GetProfileString function *141, 338*
GetProp function *81, 340*
GetRgnBox function *106, 340*
GetROP2 function *99, 341*
GetRValue function *94, 341*
GetScrollPos function *68, 341*
GetScrollRange function *68, 342*
GetStockObject function *91, 343*
GetStretchBltMode function *99, 344*
GetSubMenu function *72, 345*
GetSysColor function *74, 345, 383*
GetSysModalWindow function *345*
GetSystemDirectory function *144, 346*
GetSystemMenu function *72, 346, 691*
GetSystemMetrics function *74, 347*
GetSystemPaletteEntries function *95, 349*
GetSystemPaletteUse function *96, 349*
GetTabbedTextExtent function *112, 350*
GetTempDrive function *144, 351*
GetTempFileName function *144, 351*
GetTextAlign function *112, 352*
GetTextCharacterExtra function *354*
GetTextColor function *99, 354*
GetTextExtent function *112, 354*
GetTextFace function *112, 355*
GetTextMetrics function *112, 355*
GetThresholdEvent function *143, 356*
GetThresholdStatus function *143, 356*
GetTickCount function *43, 356*
GetTopWindow function *73, 357*
GetUpdateRect function *45, 357*
GetUpdateRgn function *45, 358*
GetVersion function *134, 359*
GetViewportExt function *100, 359*
GetViewportOrg function *100, 359*
GetWindow function *73, 360*
GetWindowDC function *45, 360*
GetWindowExt function *101, 361*
GetWindowLong function *20, 28, 361*
GetWindowOrg function *101, 362*
GetWindowRect function *42, 362*
GetWindowsDirectory function *144, 363*
GetWindowTask function *73, 363*
GetWindowText function *42, 364*
GetWindowTextLength function *42, 364*
GetWindowWord function *20, 365*
GetWinFlags function *135, 365*
GlobalAddAtom function *140, 366*
GlobalAlloc function *135, 367*
GlobalCompact function *135, 315, 368*
GlobalDeleteAtom function *140, 369*
GlobalDiscard function *135, 369*
GlobalDosAlloc function *135, 370*
GlobalDosFree function *135, 370*
GlobalFindAtom function *140, 371*
GlobalFix function *137, 371*
GlobalFlags function *135, 372*
GlobalFree function *135, 372*
GlobalGetAtomName function *140, 373*
GlobalHandle function *135, 373*
GlobalLock function *135, 374*
GlobalLRUNewest function *135, 374*

GlobalLRUOldest function *135, 375*
GlobalNotify function *135, 375*
GlobalPageLock function *137, 376*
GlobalPageUnlock function *137, 377*
GlobalReAlloc function *135, 377*
GlobalSize function *135, 379*
GlobalUnfix function *137, 379*
GlobalUnlock function *135, 380*
GlobalUnWire function *381*
GlobalUnwire function *135*
GlobalWire function *135, 381*
GMEM_DDESHARE option *367, 372*
GMEM_DISCARDABLE option *367, 372, 378*
GMEM_DISCARDED option *372*
GMEM_FIXED option *367*
GMEM_MODIFY option *378*
GMEM_MOVEABLE option *367, 378*
GMEM_NOCOMPACT option *367, 378*
GMEM_NODISCARD option *368, 378*
GMEM_NOT_BANKED option *315, 368, 372*
GMEM_NOTIFY option *368*
GMEM_ZEROINIT option *368, 379*
Graphics device interface
  defined *3*
GRAY_BRUSH stock object *343*
GrayString function *45, 382*
GW_CHILD option *360*
GW_HWNDFIRST option *360*
GW_HWNDLAST option *360*
GW_HWNDNEXT option *331, 360*
GW_HWNDPREV option *331, 360*
GW_OWNER option *360*
GWL_EXSTYLE option *362, 534*
GWL_STYLE option *362, 534*
GWL_WNDPROC option *362, 534*
GWW_HINSTANCE option *365, 545*
GWW_HWNDPARENT option *365*
GWW_ID option *365, 545*

H

Handles
  instance *24*
Help application *574*
HELP_CONTEXT option *574*
HELP_HELPONHELP option *574*
HELP_INDEX option *575*
HELP_KEY option *575*
HELP_MULTIKEY option *575*
HELP_QUIT option *575*
HELP_SETINDEX *575*
HIBYTE utility macro *143, 384*
HideCaret function *75, 385*
HiliteMenuItem function *72, 385*
HIWORD utility macro *143, 309, 386*
HOLLOW_BRUSH stock object *343*
Hook chain *224*
Hook function *224*
HORZRES device capability *305*
HORZSIZE device capability *305*
HS_BDIAGONAL brush hatch style *196*
HS_CROSS brush hatch style *196*
HS_DIAGCROSS brush hatch style *196*
HS_FDIAGONAL brush hatch style *196*
HS_HORIZONTAL brush hatch style *196*
HS_VERTICAL brush hatch style *196*
HTBOTTOM mouse-position code *673*
HTBOTTOMLEFT mouse-position code *673*
HTBOTTOMRIGHT mouse-position code *673*
HTCAPTION mouse-position code *673*
HTCLIENT mouse-position code *673*
HTERROR mouse-position code *673*
HTGROWBOX mouse-position code *673*
HTHSCROLL mouse-position code *673*
HTLEFT mouse-position code *673*
HTMENU mouse-position code *673*
HTNOWHERE mouse-position code *673*
HTREDUCE mouse-position code *673*
HTRIGHT mouse-position code *673*
HTSIZE mouse-position code *673*
HTSYSMENU mouse-position code *673*
HTTOP mouse-position code *673*
HTTOPLEFT mouse-position code *673*
HTTOPRIGHT mouse-position code *673*
HTTRANSPARENT mouse-position code *673*
HTVSCROLL mouse-position code *673*
HTZOOM mouse-position code *673*

I

IDABORT menu-item value *433*
IDC_ARROW cursor type *407*
IDC_CROSS cursor type *407*
IDC_IBEAM cursor type *407*

IDC_ICON cursor type *407*
IDC_SIZE *407*
IDC_SIZENESW cursor type *407*
IDC_SIZENS cursor type *407*
IDC_SIZENWSE cursor type *407*
IDC_SIZEWE cursor type *407*
IDC_UPARROW cursor type *407*
IDC_WAIT cursor type *407*
IDCANCEL menu-item value *273, 433*
IDI_APPLICATION icon type *408*
IDI_ASTERISK icon type *408*
IDI_EXCLAMATION icon type *408*
IDI_HAND icon type *408*
IDI_QUESTION icon type *408*
IDIGNORE menu-item value *433*
IDNO menu-item value *433*
IDOK menu-item value *273, 433*
IDRETRY menu-item value *433*
IDYES menu-item value *433*
IE_BADID error return value for OpenComm function *446*
IE_BAUDRATE error return value for OpenComm function *446*
IE_BYTESIZE error return value for OpenComm function *446*
IE_DEFAULT error return value for OpenComm function *446*
IE_HARDWARE error return value for OpenComm function *446*
IE_MEMORY error return value for OpenComm function *446*
IE_NOPEN error return value for OpenComm function *446*
IE_OPEN error return value for OpenComm function *446*
InflateRect function *83, 84, 386*
InitAtomTable function *141, 387*
InSendMessage function *14, 387*
InsertMenu function *39, 72, 198, 203, 347, 388, 691*
Integer messages *602*
Intercharacter spacing
  default *47*
Internal data structures *28*
IntersectClipRect function *107, 391*
IntersectRect function *83, 84, 392*
InvalidateRect function *45, 392*
InvalidateRgn function *45, 393*
InvertRect function *45, 394*
InvertRgn function *106, 394*
IsCharAlpha function *139, 395*
IsCharAlphaNumeric function *139, 395*
IsCharLower function *139, 395*
IsCharUpper function *139, 396*
IsChild function *73, 396*
IsClipboardFormatAvailable function *74, 396*
IsDialogMessage function *58, 397*
IsDlgButtonChecked function *58, 398*
IsIconic function *42, 398*
IsRectEmpty function *84, 398*
IsWindow function *73, 399*
IsWindowEnabled function *43, 399*
IsWindowVisible function *42, 399*
IsZoomed function *42, 400*

K

Keyboard
  using with dialog boxes *67*
KillTimer function *43, 400*

L

LB_ADDSTRING message *594, 627, 628, 629, 632, 634*
LB_DELETESTRING message *594, 627, 648*
LB_DIR message *594, 628*
LB_FINDSTRING message *594, 628*
LB_GETCARETINDEX message *629*
LB_GETCOUNT message *595, 629*
LB_GETCURSEL message *595, 629*
LB_GETHORIZONTALEXTENT message *595, 630*
LB_GETITEMDATA message *595, 630*
LB_GETITEMHEIGHT message *630*
LB_GETITEMRECT message *595, 631*
LB_GETSEL message *595, 631*
LB_GETSELCOUNT message *595, 631*
LB_GETSELITEMS message *595, 631*
LB_GETTEXT message *595, 632*
LB_GETTEXTLEN message *595, 632*
LB_GETTOPINDEX message *595, 632*
LB_INSERTSTRING message *595, 628, 629, 632, 633, 634*
LB_RESETCONTENT message *595, 633, 648*

LB_SELECTSTRING message *595, 633*
LB_SELITEMRANGE message *595, 634*
LB_SETCARETINDEX message *634*
LB_SETCOLUMNWIDTH message *214, 595, 635*
LB_SETCURSEL message *595, 635*
LB_SETHORIZONTALEXTENT message *595, 635*
LB_SETITEMDATA message *595, 630, 636*
LB_SETITEMHEIGHT message *636*
LB_SETSEL message *595, 636*
LB_SETTABSTOPS message *595, 637*
LB_SETTOPINDEX message *595, 637*
LBN_DBLCLK message *598, 638*
LBN_ERRSPACE message *598, 638*
LBN_KILLFOCUS message *598, 638*
LBN_SELCHANGE message *598, 639*
LBN_SETFOCUS message *598, 639*
LBS_EXTENDEDSEL control style *214*
LBS_HASSTRINGS control style *214, 627, 628, 629, 630, 632, 633, 634*
LBS_MULTICOLUMN control style *214, 635*
LBS_MULTIPLESEL control style *214*
LBS_NOREDRAW control style *215*
LBS_NOTIFY control style *215*
LBS_OWNERDRAWFIXED control style *215*
LBS_OWNERDRAWVARIABLE control style *215*
LBS_SORT control style *215*
LBS_STANDARD control style *215*
LimitEMSPages function *135*
LimitEmsPages function *402*
LINECAPS device capability *307*
LineDDA function *107, 402*
LineTo function *107, 403*
LISTBOX control class *208*
LMEM_DISCARDABLE option *413, 415, 418*
LMEM_DISCARDED option *415*
LMEM_FIXED option *414*
LMEM_MODIFY option *414, 418*
LMEM_MOVEABLE option *414, 418*
LMEM_NOCOMPACT option *414, 418*
LMEM_NODISCARD option *414, 419*
LMEM_ZEROINIT option *414, 419*
LoadAccelerators function *138, 404*
LoadBitmap function *110, 138, 405*
LoadCursor function *77, 138, 406*
LoadIcon function *138, 407*
LoadLibrary function *134, 408*
LoadMenu function *139, 409*
LoadMenuIndirect function *72, 410*
LoadModule function *146, 410*
LoadResource function *139, 412*
LoadString function *139, 412*
LOBYTE utility macro *143, 413*
LocalAlloc function *135, 413*
LocalCompact function *135, 414*
LocalDiscard function *136, 415*
LocalFlags function *136, 415*
LocalFree function *136, 416*
LocalHandle function *136, 416*
LocalInit function *136, 416*
LocalLock function *136, 417*
LocalReAlloc function *136, 417*
LocalShrink function *136, 419*
LocalSize function *136, 420*
LocalUnlock function *136, 420*
LockData function *136, 420*
LockResource function *139, 421*
LockSegment function *136, 137, 421*
LOGPALETTE data structure *199*
LOGPIXELSX device capability *305*
LOGPIXELSY device capability *305*
LOWORD utility macro *143, 309, 423*
LPtoDP function *105, 424*
lstrcat function *139, 425*
lstrcmp function *139, 425*
lstrcmpi function *139, 426*
lstrcpy function *139, 426*
lstrlen function *140, 427*
LTGRAY_BRUSH stock object *343*

M

MAKEINTATOM utility macro *141, 143, 428*
MAKEINTRESOURCE utility macro *143, 292, 429*
MAKELONG utility macro *143, 429*
MAKEPOINT utility macro *143, 429*
MakeProcInstance function *134, 429*
MapDialogRect function *58, 430*
Mapping mode
  default *47*
MapVirtualKey function *44, 431*

max macro *432*
MB_ABORTRETRYIGNORE option *433*
MB_APPLMODAL option *433*
MB_DEFBUTTON1 option *434*
MB_DEFBUTTON2 option *434*
MB_DEFBUTTON3 option *434*
MB_ICONASTERISK option *434*
MB_ICONEXCLAMATION option *434*
MB_ICONHAND option *434*
MB_ICONINFORMATION option *434*
MB_ICONQUESTION option *434*
MB_ICONSTOP option *434*
MB_OK option *434*
MB_OKCANCEL option *434*
MB_RETRYCANCEL option *434*
MB_SYSTEMMODAL option *434*
MB_TASKMODAL option *434*
MB_YESNO option *434*
MB_YESNOCANCEL option *434*
MEASUREITEMSTRUCT data structure *668*
Memory
  least-recently used *374, 375*
Menu
  pop-up
    described *39*
Menu bar
  described *39*
Menu functions *72*
Menu item
  removing *470*
MERGECOPY raster operation *163*
MERGEPAINT raster operation *163*
Message functions *14*
MessageBeep function *75, 432*
MessageBox function *75, 432*
Metafile functions *124*
Metafiles
  changing *127*
  creating *125*
  creating and using *125, 126*
  deleting *127*
  storing *126*
MF_BITMAP menu flag *158, 389, 436, 669*
MF_BYCOMMAND menu flag *169, 253, 386, 389, 436*
MF_BYPOSITION menu flag *170, 253, 386, 389, 436*
MF_CHECKED menu flag *158, 170, 323, 390, 437, 669*
MF_DISABLED menu flag *158, 253, 323, 390, 437, 669*
MF_ENABLED menu flag *158, 253, 323, 390, 437*
MF_GRAYED menu flag *158, 253, 323, 390, 437, 669*
MF_HILITE menu flag *386*
MF_MENUBARBREAK menu flag *158, 323, 390, 437*
MF_MENUBREAK menu flag *158, 323, 390, 437*
MF_MOUSESELECT menu flag *669*
MF_OWNERDRAW menu flag *158, 390, 437, 669*
MF_POPUP menu flag *158, 390, 437, 669*
MF_SEPARATOR menu flag *159, 323, 390, 437*
MF_STRING menu flag *159, 390, 437*
MF_SYSMENU menu flag *669*
MF_UNCHECKED menu flag *159, 170, 323, 391, 438*
MF_UNHILITE menu flag *386*
min macro *434*
MK_CONTROL mouse-key code *660, 661, 662, 663, 670, 682, 683*
MK_LBUTTON mouse-key code *660, 662, 663, 670, 682, 683*
MK_MBUTTON mouse-key code *660, 661, 662, 670, 682, 683*
MK_RBUTTON mouse-key code *660, 661, 662, 663, 670, 682*
MK_SHIFT mouse-key code *660, 661, 662, 663, 670, 682, 683*
MM_ANISOTROPIC mapping mode *504*
MM_HIENGLISH mapping mode *504*
MM_HIMETRIC mapping mode *504*
MM_ISOTROPIC mapping mode *504*
MM_LOENGLISH
  mapping mode *105*
MM_LOENGLISH mapping mode *504*
MM_LOMETRIC mapping mode *504*
MM_TEXT
  mapping mode *104*
MM_TEXT mapping mode *504*
MM_TWIPS mapping mode *504*
ModifyMenu function *72, 347, 435, 691*

MoveTo function *107, 438*
MoveWindow function *42, 438*
MSGF_DIALOGBOX filter-function message type *543, 544*
MSGF_MENU filter-function message type *543, 544*
MSGF_MESSAGEBOX filter-function message type *544*
MulDiv function *143, 439*
Multitasking
  defined *2*

N

nAccelerators
  loading or translating *17*
  with dialog boxes *67*
nBackground
  color@default *46*
  mode@default *46*
nBitBlt function
  and color palettes *99*
nBitmap
  device-dependent
    getting device-independent bits from *309*
  device-independent
    creating *189*
    displaying *498*
    retrieving bits *309*
    setting on display surface *498*
  memory
    setting bits in *497*
nBitmap functions
  device independent *111, 112*
nBrush
  alignment *53*
  creating *179*
  default *46*
nCaret
  creating and displaying *76*
  functions *75*
  sharing *76*
nChangeMenu *See also* AppendMenu, *See also* DeleteMenu, *See also* InsertMenu, *See also* ModifyMenu, *See also* RemoveMenu
nCharacter
  determining if alphabetic *395*
  determining if alphanumeric *395*
  determining if lowercase *395*
  determining if uppercase *396*
nCheckmark
  custom *506*
  getting size of *321*
nChild window
  described *35*
nClass
  functions@default messages *33*
  functions@defining *30, 33*
  functions@examining *30, 33*
  functions@receiving *30, 33*
  functions@responding *30, 33*
  messages@declaring *33*
  messages@sending *33*
  messages@values *33*
  registering *33*
  styles@child *35*
  styles@overlapped *34*
  styles@owned *35*
  styles@pop-up *35*
  window
    unregistering *567*
nClasses
  Application Global *21*
  Application Local *21*
  class background brush@assigning *26*
  class background brush@setting *26*
  class name@assigning *24*
  class name@global uniqueness *24*
  creating *20*
  Cursor *25*
  defining and registering *20*
  determining ownership *22*
  display contexts *30*
  elements *23*
  elements@assigning *23*
  elements@class names *23*
  instance handle *24*
  predefined *22*
  redrawing client area *29*
  registering *22*
  shared *22*
  styles *27*
  System Global *21*

nClient area
child window *35*
redrawing *29*
update region *52*
nClipboard
functions *74*
nColor
logical-palette index *450*
using color in logical palette *450, 451*
nColor palettes
updating client area *568*
nCombo box
owner-draw *65*
owner-draw@sorting *645*
nCOMBOBOX control class
control styles *212*
nContexts
class and private *30*
classes
displaying *47*
displaying *46*
displaying cache *50*
displaying common defaults *47*
painting changes *50*
private display *48*
window display *49*
WM_PAINT message *50*
nControl
current font *653*
owner-draw
drawing *650, 668*
measuring *668*
setting current font *685*
nControl class
COMBOBOX@control styles *212*
COMBOBOX@described *208*
nControl styles
COMBOBOX class *212*
nCreateWindow function
creating a child window *36*
creating an overlapped window *35*
described *19, 205*
nCursor
class *25*
confining *78*
creating custom *78*
displaying and hiding *77*
functions *76*
positioning *78*
with pointing devices *77*
nDELETEITEMSTRUCT data structure
as parameter of WM_DELETEITEM message *648*
nDestroyWindow function
described *231*
destroying modeless dialog boxes *59*
effect *41*
nDevice context
attributes and functions *88*
creating
saving and deleting *90*
nDevice driver
device capabilities *232*
nDialog box
accelerators *67*
buttons *64*
control identifiers *62*
control styles *63*
controls *62, 66*
control messages *66*
creating *59, 60*
described *57*
dimensions *66*
edit controls *64*
input function *60*
keyboard input *67*
keyboard interface@actions *66*
keyboard interface@filtering
measurements *67*
modal@creating *60, 239, 240*
modal@moveable *60*
modeless@creating *187, 188*
modeless@deleting *59*
modeless@using *59*
private window class default function *220*
redrawing *66*
return values *61*
scrolling *68*
template *60*
using *59*
nDialog boxes
keyboard interface@scrolling *67*
owner draw *66*
nDIB Bitmap *See* device-independent

nDIB_PAL_COLORS
device-independent bitmap color table option *189, 190, 309, 497, 499, 553*
nDIB_RGB_COLORS
device-independent bitmap color table option *189, 190, 309, 497, 499, 553*
nDocument conventions
\bcB\ecbold text\bcD\ec *8*
\bcF105M\ecmonospaced type\bcF255D\ec *9*
\bcMI\ecitalic text\bcD\ec *9*
curly braces ({ }) *9*
double brackets ([[ ]]) *9*
horizontal ellipses ... *9*
parentheses ( ) *8*
quotation marks (\bc169\ec \bc170\ ec)[quotation marks ( )] *9*
small capital letters *9*
vertical bar (|) *9*
vertical ellipses *9*
nDrawing
formatted text *54*
gray text *57*
icons *54*
mode
default *47*
nDRAWITEMSTRUCT
as parameter of WM_DRAWITEM message *650*
nEdit control
tab stops in *622*
nEllipses
horizontal
as document convention *9*
vertical
as document convention *9*
NETBIOS interrupt
function request (5CH) *440*
NetBIOSCall function *137, 440*
nExtents
viewport and window default *47*
nFile
closing *401*
creating *401*
help@displaying *574*
initialization@application-specific *335, 336, 578*
opening *422*
positioning the pointer *404*
reading *424*
writing *427*
nFilling mode
ALTERNATE *202, 510*
WINDING *202, 510*
nFont
average character width *119*
control
current *653*
default *47*
logical
creating *193, 195*
maximum character width *119*
pitch *119*
setting in control *685*
nFont mapping
characteristics *121, 122*
nFonts
character sets *117*
vendor specific *118*
character sets@ANSI *118*
character sets@OEM *118*
character sets@printer *118*
digitized aspect *119*
leading *117*
overhang *119*
nFunction
coordinates *108*
main loop *16*
window *18*
nFunctions
additional *83*
bitmap *110, 111, 112*
bounding rectangles *110*
caret *75, 76*
clipboard *74*
displaying *42*
drawing tools *91*
error *74*
filters *79*
hardware *43*
hook *79*
information *73*
input *43*
mapping drawing attributes *100*

menu *72*
movement *42*
obtaining device information *91*
painting *44*
property lists *80, 82*
rectangle@coordinates *83*
rectangle@specifying *83*
regions *106*
system *73*
nGDI Functions
brushes
predefined *92*
color palettes *95*
drawing-attribute functions *99*
drawing attribute functions@background mode and color *100*
drawing attribute functions@mapping funtions *100*
drawing attribute functions@stretch mode and text color *100*
drawing tool functions *91*
mapping functions *101*
mapping modes
constraining *102, 103*
transformation equations *103*
obtaining device information *91*
pens
predefined *93*
selecting fonts *120*
using drawing tools *92*
working with color palettes *96*
nHandle
task
obtaining *302*
nIcon
class *25*
drawing *54*
nInitialization file
application-specific
getting integer from *335*
getting string from *336*
writing to *578*
nInterrupt
function request (21H) *245*
function request (5CH) *440*
nKey
getting name *319*
nLine output functions
pen styles
colors and widths *108*
nLine-output functions
coordinates *108*
nList box
directory listings *65*
horizontal scrolling *630, 635*
owner-draw
described *65*
owner-draw@deleted item *648*
owner-draw@sorting *645*
tab stops in *637*
nLogical palette
and input focus *681*
changed *679*
changing entries in *508*
creating *199*
finding color in *329*
index specifier (direct) *450*
index specifier (indirect) *451*
realizing *465, 679*
selecting *483*
nMDICREATESTRUCT
as parameter of WM_MDICREATE message *664*
nMenu
changing *157, 388, 435*
class *26*
creating *198*
deleting *228*
pop-up *202*
nMenu checkmark
custom *506*
getting size of *321*
nMessage
posting to task windows *458*
nMessages
application queue *15*
bypassing the queue *15*
checking the queue *17*
clipboard *591*
closing *41*
contents *601*
described *18*
destroy message *41*
dispatching *15, 16*

examining@checking queues
passing
posting *18*
examining@formatted and transmitting *18*
generated by applications *15*
generating or processing@input events and application queue *16*
generating or processing@queuing and virtual-key *16*
input events *15*
integer *602*
keyboard input *16*
peeking *17*
posting *18*
pulling *15*
pushing *16*
ranges *602*
reading *15*
reading@without pulling *17*
reserved *602*
sending *18*
special actions *15*
string *602*
translating *16*
translating@accelerator keys *17*
translating@loops *17*
virtual keys *16*
window
default processing *227*
window functions *15*
nMessages notification *See* Notification codes
nMetafile functions
additional escapes *131*
environment *131*
information escapes *130*
printer escapes
banding *129, 130*
starting and ending *130*
terminating *130*
nMouse cursor *See* Cursor
nMultiple Document Interface (MDI)
child window@activating *663, 666*
child window@active *665*
child window@cascading *664*
child window@closing *665*
child window@creating *664*
child window@default function *225*
child window@maximizing *666*
child window@restoring *667*
child window@system accelerator *562*
child window@tiling *667*
client window *663, 664, 665, 666, 667*
frame window default function *223*
messages *600*
system accelerator *562*
nNotification codes
button *597*
edit control *597*
nOrigin
brush
default *46*
viewport
default *47*
window
default *47*
nOwner-draw control
described *65*
nPainting
functions *44*
inverting
drawing
filling *53*
rectangles *53*
systems display *44*
updating background *52*
updating displays *51*
updating nonclient area *57*
validating rectangle *570*
validating region *570*
nPalette
system
retrieving entries *349*
nPen
creating *200, 201*
position
default *46*
nPop-up menu
creating *202*
described *39*
nPrinter
initialization *271*
nPrinter functions
banding *129*
creating output *129*

nProperty list functions
  adding entries *82*
  creating *82*
  dumping contents *82*
nQueue
  application *15*
  checking *17*
nRaster fonts
  digitized aspect *119*
nRealize *See* logical palette
nRectangle functions
  additional functions *83*
  coordinates *83*
  in Windows *83*
  InflateRect *84*
  IntersectRect *84*
  IsRectEmpty *84*
  OffsetRect *84*
  PtInRect *84*
  SetRect *84*
  specifying *83*
  UnionRect *85*
nRegion
  rounded rectangle
    creating *204*
  validating *570*
nResources
  managing hooks *79*
nScrolling
  functions@controlling *69*
  functions@described *68*
  functions@processing *71*
  functions@requests *70*
  hiding *71*
  using thumb *70*
nSendMessage function
  Message deadlock caused by *18*
nSetWindowPos *219*
nStretchBlt function
  and color palettes *99*
nStrings
  comparing *425, 426*
  concatenating *425*
  copying *427*
  determining length of *427*
  formatting *579, 581*
nStyles
  dialog box controls *63*
  formatted text *55*
nSystem palette
  retrieving entries *349*
nTask
  handle
    obtaining *302*
nTask windows
  enumerating *265*
  posting messages to *458*
nTasks
  yielding control *583*
nText
  drawing *57*
  graying *56*
NULL_BRUSH stock object *343*
NULL_PEN stock object *343*
NULLREGION region type *176, 270, 298, 358, 391, 442, 443, 481*
NUMBRUSHES device capability *305*
NUMCOLORS device capability *305*
NUMFONTS device capability *305*
NUMPENS device capability *305*
nWindow
  background *52*
  background brush *23*
  brush alignment *53*
  child@close box *38*
  child@described *35*
  child@ID *36*
  child@input *36*
  child@messages *36*
  child@overlapping *37*
  child@owner window *36*
  child@showing *36*
  class@attributes *23*
  class@background brush *23*
  class@cursor
    icon
      attributes *23*
  class@described *20*
  class@functions *23*
  class@instance handle *23*
  class@menu
    styles *23*
  class@name *23*

creating *218*
dialog box *57*
function role *18*
icon *34*
main
creating *41*
open *34*
overlapped *34*
overlapping *35*
owner
describing *36*
painting rectangles *53*
pop-up@creating and showing *35*
scroll bars *38*
state *40*
styles *34*
styles@child *34, 35*
styles@owned *35*
styles@pop-up *35*
styles@state *40*
subclassing *28, 490, 534*
System menu box *38*
title bar *38*
nWindow applications
application queue *15*
dispatching messages *16*
pulling messages *15*
pushing messages *16*
reading messages *15*
yielding control *15*
nWindow class
background brush *25*
unregistering *567*
nWindow function
address *24*
receiving messages *16*
nWindows
displaying functions *42*
enumerating for a task *265*
painting@drawing *53*
painting@filling *53*
painting@inverting *53*
posting messages to a task *458*
subclassing *28*
nWindows Classes
class menu *26*
locating *21*
Window-Function address *24*
nWM_COMMAND notification codes *See* Notification codes

O

OEM_FIXED_FONT stock object *343*
OemKeyScan function *44, 440*
OemToAnsi function *140, 441*
OemToAnsiBuff function *140, 442*
OF_CANCEL option *447*
OF_CREATE option *447*
OF_DELETE option *447*
OF_EXIST option *447*
OF_PARSE option *447*
OF_PROMPT option *447*
OF_READ option *422, 447*
OF_READWRITE option *422*
OF_REOPEN option *447*
OF_VERIFY option *448*
OF_WRITE option *423, 448*
OffsetClipRgn function *107, 442*
OffsetRect function *83, 84, 443*
OffsetRgn function *106, 443*
OffsetViewportOrg function *101, 444*
OffsetWindowOrg function *101, 444*
OPAQUE background mode *487*
OpenClipboard function *74, 445*
OpenComm function *142, 445*
OpenFile function *144, 446*
OpenIcon function *42, 449*
OpenSound function *143, 449*
OutputDebugString function *145, 449*
Overlapped window *34*
Owner-draw dialog box controls *66*
OWNERDRAWFIXED resource option *668*

P

PaintRgn function *106, 450*
PALETTEINDEX utility macro *144, 450*
PALETTERGB *144*
PALETTERGB utility macro *144, 450*
palettes
resizing *473*
Parentheses ( )
as document convention *8*
PatBlt function *110, 451*

PATCOPY raster operation *163*
PATINVERT raster operation *164*
PATPAINT raster operation *164*
PC_RESERVED palette-entry option *153*
PDEVICESIZE device capability *306*
PeekMessage function *14, 452*
Pie function *109, 110, 454*
PLANES device capability *305*
PlayMetaFile function *124, 455*
PlayMetaFileRecord function *124, 455*
PM_NOREMOVE option *453*
PM_NOYIELD option *453*
PM_REMOVE option *453*
POINT data structure *234*
Polygon-filling mode
default *47*
Polygon function *109, 456*
POLYGONALCAPS device capability *307*
Polyline function *107, 456*
PolyPolygon function *109, 457*
PostAppMessage function *14, 458*
PostMessage function *14, 458*
PostQuitMessage function *14, 459*
Printer-control functions *128*
Printer device driver capabilities *232*
ProfClear function *145, 459*
ProfFinish function *145, 460*
ProfFlush function *145, 460*
ProfInsChk function *145, 460*
ProfSampRate function *145, 461*
ProfSetup function *145, 460, 462*
ProfStart function *145, 462*
ProfStop function *145, 462*
Property list functions *80*
PtInRect function *83, 84, 463*
PtInRegion function *106, 463*
PtVisible function *107, 463*

Q

Quotation marks (\bc169\ec \bc170\ec)
as document convention[Quotation marks ( ),
as document convention] *9*

R

R2_MASKNOTPEN raster drawing mode *514*
R2_MASKPEN raster drawing mode *515*
R2_MASKPENNOT raster drawing mode *514*
R2_MERGENOTPEN raster drawing mode *514*
R2_MERGEPEN raster drawing mode *514*
R2_MERGEPENNOT raster drawing mode *514*
R2_NOT raster drawing mode *514*
R2_NOTCOPYPEN raster drawing mode *514*
R2_NOTMASKPEN raster drawing mode *515*
R2_NOTMERGEPEN raster drawing mode *514*
R2_NOTXORPEN raster drawing mode *515*
R2_XORPEN raster drawing mode *515*
RASTERCAPS device capability *306*
RC_BANDING device capability *306*
RC_BITBLT device capability *306*
RC_BITMAP64 device capability *306*
RC_DI_BITMAP device capability *306*
RC_DIBTODEV device capability *306*
RC_FLOODFILL device capability *306*
RC_GDI20_OUTPUT device capability *306*
RC_PALETTE device capability *306*
RC_SCALING device capability *306*
RC_STRETCHBLT device capability *306*
RC_STRETCHDIB defice capability *306*
ReadComm function *142, 464*
RealizePalette function *96, 465*
Rectangle
validating *570*
Rectangle function *109, 465*
RectInRegion function *106, 466*
RectVisible function *107, 466*
Region functions *106*
RegisterClass function *20, 467*
RegisterClipboardFormat *74*
RegisterClipboardFormat function *468*
RegisterWindowMessage function *468*
Relative-absolute flag
default setting[Relative absolute flag
default setting] *47*
ReleaseCapture function *43, 469*
ReleaseDC function *45, 469*
RemoveFontResource function *113, 470*
RemoveMenu function *72, 470*
RemoveProp function *81, 471*
ReplyMessage function *14, 472*
Reserved messages *602*
RESETDEV communication function code *269*
ResizePalette function *473*
RestoreDC function *88, 473*

RGB utility macro *144, 474*
RGN_AND region-combining mode *175*
RGN_COPY region-combining mode *175*
RGN_DIFF region-combining mode *176*
RGN_OR region-combining mode *176*
RGN_XOR region-combining mode *176*
RoundRect function *109, 474*
RT_ACCELERATOR resource type *279*
RT_BITMAP resource type *279*
RT_DIALOG resource type *279*
RT_FONT resource type *279*
RT_MENU resource type *279*
RT_RCDATA resource type *279*

S

S_ALLTHRESHOLD voice-queue state *572*
S_LEGATO voice note style *529*
S_NORMAL voice note style *529*
S_PERIOD1024 voice frequency *517*
S_PERIOD2048 voice frequency *517*
S_PERIOD512 voice frequency *517*
S_PERIODVOICE voice frequency *517*
S_QUEUEEMPTY voice-queue state *572*
S_SERDCC voice error code *530*
S_SERDDR voice error code *532*
S_SERDFQ voice error code *532*
S_SERDLN voice error code *530*
S_SERDMD voice error code *529*
S_SERDNT voice error code *530*
S_SERDRC voice error code *530*
S_SERDSH voice error code *530*
S_SERDTP voice error code *529*
S_SERDVL voice error code *529, 532*
S_SERMACT voice error code *531*
S_SEROFM voice error code *531*
S_SERQFUL voice error code *529, 530, 532*
S_STACCATO voice note style *529*
S_THRESHOLD voice queue status *572*
S_WHITE1024 voice frequency *517*
S_WHITE2048 voice frequency *517*
S_WHITE512 voice frequency *517*
S_WHITEVOICE voice frequency *517*
SaveDC function *88, 476*
SB_BOTH scroll-bar type *548*
SB_BOTTOM scrolling request *655, 656, 696*
SB_CTL scroll-bar type *342, 515, 516, 548*
SB_ENDSCROLL scrolling request *655, 656, 696*
SB_HORZ scroll-bar type *342, 515, 516, 548*
SB_LINEDOWN scrolling request *655, 656, 696*
SB_LINEUP scrolling request *655, 656, 696*
SB_PAGEDOWN scrolling request *655, 656, 696*
SB_PAGEUP scrolling request *655, 656, 696*
SB_THUMBPOSITION scrolling request *655, 656, 696, 697*
SB_THUMBTRACK scrolling request *655, 696*
SB_TOP scrolling request *655, 656, 696, 697*
SB_VERT scroll-bar type *342, 343, 515, 516, 548*
SBS_BOTTOMALIGN control style *215*
SBS_HORZ control style *215*
SBS_LEFTALIGN control style *216*
SBS_RIGHTALIGN control style *216*
SBS_SIZEBOX control style *216*
SBS_SIZEBOXBOTTOMRIGHTALIGN control style *216*
SBS_SIZEBOXTOPLEFTALIGN control style *216*
SBS_TOPALIGN control style *216*
SBS_VERT control style *216*
SC_CLOSE system command *690*
SC_HOTKEY system command *690*
SC_HSCROLL system command *690*
SC_KEYMENU system command *690*
SC_MAXIMIZE system command *690*
SC_MINIMIZE system command *690*
SC_MOUSEMENU system command *690*
SC_MOVE system command *690*
SC_NEXTWINDOW system command *690*
SC_PREVWINDOW system command *690*
SC_RESTORE system command *690*
SC_SCREENSAVE system command *690*
SC_SIZE system command *690*
SC_TASKLIST *690*
SC_VSCROLL system command *690*
ScaleViewportExt function *101, 476*
ScaleWindowExt function *101, 477*
ScreenToClient function *106, 477*
Scroll bars *38*
SCROLLBAR control class *209*
ScrollDC function *68, 478*
Scrolling *71*

ScrollWindow function *68, 479*
SelectClipRgn function *107, 480*
SelectObject function *91, 481*
SelectPalette function *96, 483*
SendDlgItemMessage function *58, 483*
SendMessage function *14, 15, 484*
SetActiveWindow function *43, 485*
SetBitmapBits function *110, 485*
SetBitmapDimension function *111, 486*
SetBkColor function *99, 486*
SetBkMode function *99, 487*
SetBrushOrg function *91, 487*
SetCapture function *43, 488*
SetCaretBlinkTime function *75, 488*
SetCaretPos function *75, 488*
SetClassLong function *20, 29, 489*
SetClassWord function *20, 490*
SetClipboardData function *74, 491*
SetClipboardViewer function *74, 493*
SetCommBreak function *142, 494*
SetCommEventMask function *142, 494*
SetCommState function *142, 495*
SetCursor function *77, 495*
SetCursorPos function *77, 496*
SetDIBits function *99, 111, 306, 496, 497*
SetDIBitsToDevice function *111, 306, 498, 499*
SetDlgItemInt function *58, 499*
SetDlgItemText function *59, 500*
SetDoubleClickTime function *43, 500*
SETDTR communication function code *269*
SetEnvironment function *131, 501*
SetErrorMode function *138, 501*
SetFocus function *43, 502*
SetHandleCount function *144, 502*
SetKeyboardState function *44, 503*
SetMapMode function *101, 503*
SetMapperFlags function *113, 119, 505*
SetMenu function *72, 505*
SetMenuItemBitmaps function *72, 159, 321, 506*
SetMessageQueue function *14, 507*
SetMetaFileBits function *124, 507*
SetPaletteEntries function *96, 508*
SetParent function *73, 508*
SetPixel function *111, 509*
SetPolyFillMode function *99, 456, 457, 509*
SetProp function *81, 510*
SetRect function *84, 511*
SetRectEmpty function *83, 511*
SetRectRgn function *106, 512*
SetResourceHandler function *139, 512*
SetROP2 function *99, 514*
SETRTS communication function code *269*
SetScrollPos function *68, 515*
SetScrollRange function *68, 516*
SetSoundNoise function *143, 516*
SetStretchBltMode function *99, 517*
SetSwapAreaSize function *136, 518*
SetSysColors function *74, 519*
SetSysModalWindow function *43, 520*
SetSystemPaletteUse function *96, 99, 349, 520*
SetTextAlign fulnction *522*
SetTextAlign function *112*
SetTextCharacterExtra function *523*
SetTextColor function *99, 383, 523*
SetTextJustification function *112, 524*
SetTimer function *43, 525*
SetViewportExt function *101, 526*
SetViewportOrg function *101, 527*
SetVoiceAccent function *143, 528*
SetVoiceEnvelope function *143, 529*
SetVoiceNote function *143, 530*
SetVoiceQueueSize function *143, 531*
SetVoiceSound function *143, 531*
SetVoiceThreshold function *143, 532*
SetWindowExt function *101, 532*
SetWindowLong function *20, 28, 533*
SetWindowOrg function *101, 534*
SetWindowPos function *42, 535*
SetWindowsHook function *79, 536*
SetWindowText function *38, 42, 545*
SetWindowWord function *20, 545*
SETXOFF communication function code *269*
SETXON communication function code *269*
ShowCaret function *75, 546*
ShowCursor function *77, 546*
ShowOwnedPopups function *42, 547*
ShowScrollBar function *68, 547*
ShowWindow function *42, 411, 548, 573*
SIMPLEREGION region type *176, 270, 298, 358, 391, 442, 443, 481*
SIZEFULLSCREEN window-sizing request *687*
SIZEICONIC window-sizing request *687*
SIZENORMAL window-sizing request *687*

SizeofResource function *139, 549*
SIZEZOOMHIDE window-sizing request *687*
SIZEZOOMSHOW window-sizing request *687*
SM_CXBORDER system-metric value *348*
SM_CXDLGFRAME system-metric value *348*
SM_CXFRAME system-metric value *348*
SM_CXFULLSCREEN system-metric value *348*
SM_CXHSCROLL system-metric value *348*
SM_CXHTHUMB system-metric value *348*
SM_CXMINTRACK system-metric value *348*
SM_CXSIZE system-metric value *348*
SM_CXVSCROLL system-metric value *348*
SM_CYBORDER system-metric value *348*
SM_CYDLGFRAME system-metric value *348*
SM_CYFRAME system-metric value *348*
SM_CYFULLSCREEN system-metric value *348*
SM_CYHSCROLL system-metric value *348*
SM_CYSIZE system-metric value *348*
SM_CYVSCROLL system-metric value *348*
SM_CYVTHUMB system-metric value *348*
SM_DEBUG system-metric value *349*
SM_MOUSEPRESENT system-metric value *348*
SM_SWAPBUTTON system-metric value *349*
Small capital letters
  as document convention *9*
SP_ERROR escape error code *268*
SP_OUTOFDISK escape error code *268*
SP_OUTOFMEMORY escape error code *268*
SP_USERABORT escape error code *268*
SRCAND raster operation *164*
SRCCOPY raster operation *163, 164*
SRCERASE raster operation *163, 164*
SRCINVERT raster operation *164*
SRCPAINT raster operation *164*
SS_BLACKFRAME control style *216*
SS_BLACKRECT control style *216*
SS_CENTER control style *217*
SS_GRAYFRAME control style *217*
SS_GRAYRECT control style *217*
SS_ICON control style *217*
SS_LEFT control style *217*
SS_LEFTNOWORDWRAP control style *217*
SS_NOPREFIX control style *217*
SS_RIGHT control style *217*
SS_SIMPLE control style *218*
SS_USERITEM control style *218*
SS_WHITEFRAME control style *218*
SS_WHITERECT control style *218*
StartSound function *143, 549*
STATIC control class *209*
StopSound function *143, 550*
StretchBlt function *111, 550*
StretchDIBits function *111, 552*
Stretching mode
  default *47*
String messages *602*
Subclassing windows *28, 490, 534*
SW_HIDE window state *548*
SW_MINIMIZE window state *548*
SW_PARENTCLOSING window state *687*
SW_PARENTOPENING window state *687*
SW_RESTORE window state *548*
SW_SHOW window state *549*
SW_SHOWMAXIMIZED window state *549*
SW_SHOWMINIMIZED window state *549*
SW_SHOWMINNOACTIVE window state *549*
SW_SHOWNA window state *549*
SW_SHOWNOACTIVATE window state *549*
SW_SHOWNORMAL window state *549*
SwapMouseButton function *43, 554*
SwapRecording function *145, 554*
SwitchStackBack function *136, 555*
SwitchStackTo function *136, 555*
SWP_DRAWFRAME window-position flag *222, 536*
SWP_HIDEWINDOW window-position flag *222, 536*
SWP_NOACTIVATE window-position flag *222, 536*
SWP_NOMOVE window-position flag *222, 536*
SWP_NOREDRAW window-position flag *222, 536*
SWP_NOSIZE window-position flag *222, 536*
SWP_NOZORDER window-position flag *222, 536*
SWP_SHOWWINDOW window-position flag *222, 536*
SyncAllVoices function *143, 556*
System
  functions *73*
System accelerator (MDI) *562*
SYSTEM_FIXED_FONT stock object *344*
SYSTEM_FONT stock object *344*
System menu box *38*

System services interface
  defined *4*
Systems display
  painting
    functions *44*

T

TA_BASELINE text-alignment flag *353, 522*
TA_BOTTOM text-alignment flag *353, 522*
TA_CENTER text-alignment flag *353, 522*
TA_LEFT text-alignment flag *353, 522*
TA_NOUPDATECP text-alignment flag *353, 522*
TA_RIGHT text-alignment flag *353, 522*
TA_TOP text-alignment flag *353, 523*
TA_UPDATECP text-alignment flag *353, 523*
TabbedTextOut function *112, 556*
Tasks
  yielding control *15*
TECHNOLOGY device capability *305*
Text color
  default *47*
Text functions *112*
TEXTCAPS device capability *307*
TextOut function *112, 557*
Throw function *138, 558*
Timer
  killing *400*
Title bar *38*
ToAscii function *140, 559*
TPM_RIGHTBUTTON pop-up menu flag *560*
TrackPopupMenu function *39, 72, 203, 560*
TranslateAccelerator function *14, 17, 560*
TranslateMDISysAccel function *14, 562*
TranslateMessage function *14, 16, 562*
TransmitCommChar function *142, 563*
TRANSPARENT background mode *487*

U

UngetCommChar function *142, 563*
UnhookWindowsHook function *79, 564*
UnionRect function *83, 85, 565*
UnlockData function *136, 565*
UnlockResource function *139, 565*
UnLockSegment function *136*
UnlockSegment function *137, 566*
UnrealizeObject function *91, 566*
UnregisterClass function *20, 567*
UpdateColors function *96, 568*
UpdateWindow function *45, 568*
Updating region
  client area *52*

V

ValidateCodeSegments function *145, 568*
ValidateFreeSpaces function *145, 569*
ValidateRect function *45, 569*
ValidateRgn function *45, 570*
Variable-pitch font attribute *119*
Vertical bar (|)
  as document convention *9*
VERTRES device capability *305*
VERTSIZE device capability *305*
Viewport extents
  default *47*
Viewport origin
  default *47*
Virtual keys *16*
VkKeyScan function *44, 570*

W

WaitMessage function *14, 571*
WaitSoundState function *143, 572*
WH_CALLWNDPROC windows-hook type *537, 564*
WH_GETMESSAGE windows-hook type *537, 564*
WH_JOURNALPLAYBACK windows-hook type *537, 564*
WH_JOURNALRECORD windows-hook type *537, 564*
WH_KEYBOARD windows-hook type *80, 537, 564*
WH_MSGFILTER windows-hook type *80, 537, 564*
WH_SYSMSGFILTER windows-hook type *537*
WHITE_BRUSH stock object *343*
WHITE_PEN stock object *343*
WHITENESS raster-operation code *164*
WHITEONBLACK stretching mode *518*
WINDING filling mode *202, 510*
WINDING polygon-filling mode *202, 334, 509*

Window bar menu *39*
Window extents
  default *47*
Window manager interface
  defined *2*
Window origin
  default *47*
WindowFromPoint function *73, 106, 572*
WinExec function *146, 573*
WinHelp function *146, 574*
WinMain function
  main loop *16, 17*
WM_ACTIVATE message *588, 639*
WM_ACTIVATEAPP message *588, 640*
WM_ASKCBFORMATNAME message *591, 640*
WM_CANCELMODE message *588, 641*
WM_CHANGECBCHAIN message *591, 641*
WM_CHAR message *590, 641*
WM_CHARTOITEM message *590, 642*
WM_CHILDACTIVATE message *588, 643*
WM_CLEAR message *594, 643*
WM_CLOSE message *588, 643*
WM_CLOSE message message *41*
WM_COMMAND message *590, 644*
WM_COMPACTING message *592, 644*
WM_COMPAREITEM message *596, 645*
WM_COPY *594, 645*
WM_CREATE message *219, 588, 646*
WM_CTLCOLOR message *588, 646*
WM_CUT message *594, 647*
WM_DEADCHAR message *590, 647*
WM_DELETEITEM message *596, 607, 611, 628, 633, 648*
WM_DESTROY message *41, 588, 648*
WM_DESTROYCLIPBOARD message *591, 649*
WM_DEVMODECHANGE message *592, 649*
WM_DRAWCLIPBOARD message *591, 649*
WM_DRAWITEM message *596, 650*
WM_ENABLE message *588, 650*
WM_ENDSESSION message *588, 650*
WM_ENTERIDLE message *588, 651*
WM_ERASEBKGND message *588, 651*
WM_FONTCHANGE message *592, 652*
WM_GETDLGCODE message *588, 652*
WM_GETFONT message *592, 653*
WM_GETMINMAXINFO message *588, 653*
WM_GETTEXT message *588, 654*
WM_GETTEXTLENGTH message *588, 654*
WM_HSCROLL message *38, 590, 598, 655*
WM_HSCROLLCLIPBOARD message *591, 656*
WM_ICONERASEBKGND message *588, 656*
WM_INITDIALOG message *187, 188, 239, 240, 589, 657, 668*
WM_INITMENU message *589, 657*
WM_INITMENUPOPUP message *589, 658*
WM_KEYDOWN message *590, 658*
WM_KEYUP message *590, 659*
WM_KILLFOCUS message *588, 660*
WM_LBUTTONDBLCLK message *590, 660*
WM_LBUTTONDOWN message *590, 661*
WM_LBUTTONUP message *590, 661*
WM_MBUTTONDBLCLK message *590, 662*
WM_MBUTTONDOWN message *590, 662*
WM_MBUTTONUP message *590, 663*
WM_MDIACTIVATE message *600, 663, 664*
WM_MDICASCADE message *600, 664*
WM_MDICREATE message *600, 664*
WM_MDIDESTROY message *600, 665*
WM_MDIGETACTIVE message *600, 665*
WM_MDIICONARRANGE message *600, 666*
WM_MDIMAXIMIZE message *600, 666*
WM_MDINEXT message *600, 666*
WM_MDIRESTORE message *600, 667*
WM_MDISETMENU message *600, 667*
WM_MDITILE message *600, 667*
WM_MEASUREITEM message *596, 668*
WM_MENUCHAR message *588, 668*
WM_MENUSELECT message *588, 669*
WM_MOUSEACTIVATE message *590, 669*
WM_MOUSEMOVE message *590, 670*
WM_MOVE message *588, 671*
WM_NCACTIVATE message *599, 664, 671*
WM_NCCALCSIZE message *599, 671*
WM_NCCREATE message *599, 672*
WM_NCDESTROY message *599, 672*
WM_NCHITTEST message *599, 672*
WM_NCLBUTTONDBLCLK message *599, 673*
WM_NCLBUTTONDOWN message *599, 674*
WM_NCLBUTTONUP message *599, 674*
WM_NCMBUTTONDBLCLK message *599, 674*
WM_NCMBUTTONDOWN message *599, 675*
WM_NCMBUTTONUP message *599, 675*
WM_NCMOUSEMOVE message *599, 675*

WM_NCPAINT message 599, 676
WM_NCRBUTTONDBLCLK message 599, 676
WM_NCRBUTTONDOWN message 599, 676
WM_NCRBUTTONUP message 599, 677
WM_NEXTDLGCTL message 592, 677
WM_PAINT message 51, 588, 677
WM_PAINTCLIPBOARD message 591, 678
WM_PAINTICON message 589, 678
WM_PALETTECHANGED message 592, 679
WM_PARENTNOTIFY message 589, 679
WM_PASTE message 594, 680
WM_QUERYDRAGICON function 680
WM_QUERYDRAGICON message 589
WM_QUERYENDSESSION message 589, 681
WM_QUERYNEWPALETTE message 589, 681
WM_QUERYOPEN message 589, 681
WM_QUIT message 42, 589, 682
WM_RBUTTONDBLCLK message 590, 682
WM_RBUTTONDOWN message 590, 682
WM_RBUTTONUP message 590, 683
WM_RENDERALLFORMATS message 591, 683
WM_RENDERFORMAT message 591, 684
WM_SETCURSOR message 590, 684
WM_SETFOCUS message 589, 684
WM_SETFONT message 589, 592, 685
WM_SETREDRAW message 589, 685
WM_SETTEXT message 589, 686
WM_SHOWWINDOW message 589, 686
WM_SIZE message 589, 687
WM_SIZECLIPBOARD message 591, 687
WM_SPOOLERSTATUS message 592, 688
WM_SYSCHAR message 591, 688
WM_SYSCOLORCHANGE message 592, 689
WM_SYSCOMMAND message 591, 690
WM_SYSDEADCHAR message 591, 691
WM_SYSKEYDOWN message 591, 691
WM_SYSKEYUP message 591, 693
WM_TIMECHANGE message 592, 694
WM_TIMER message 590, 694
WM_UNDO message 594, 695
WM_USER message 602
WM_VKEYTOITEM message 590, 695
WM_VSCROLL message 38, 590, 598, 695
WM_VSCROLLCLIPBOARD message 591, 696
WM_WININICHANGE message 592, 697
WNDCLASS data structure 292
WriteComm function 142, 576
WritePrivateProfileString function 141, 577
WriteProfileString function 141, 578
WS_BORDER window style 209
WS_CAPTION window style 38, 60, 209, 664
WS_CHILD window style 36, 209, 664
WS_CHILDWINDOW window style 209
WS_CLIPCHILDREN window style 210, 664
WS_CLIPSIBLINGS window style 210, 664
WS_DISABLED window style 210
WS_DLGFRAME window style 210
WS_EX_DLGMODALFRAME extended window style 219
WS_EX_NOPARENTNOTIFY extended window style 219
WS_EX_TOPMOST extended window style 219
WS_GROUP control style 210
WS_HSCROLL window style 210
WS_ICONIC window style 210
WS_MAXIMIZE window style 210
WS_MAXIMIZEBOX window style 210, 664
WS_MINIMIZE window style 210
WS_MINIMIZEBOX window style 210
WS_OVERLAPPED window style 35, 210
WS_OVERLAPPEDWINDOW window style 35, 210
WS_POPUP window style 35, 210
WS_POPUPWINDOW window style 210
WS_SYSMENU window style 38, 60, 209, 211, 664
WS_TABSTOP window style 211
WS_THICKFRAME window style 211, 664
WS_VISIBLE window style 211
WS_VSCROLL window style 211
wsprintf function 140, 579
wvsprintf function 140, 581, 582

Y

Yield function 138, 583

WINDOWS API

VOLUME I

BORLAND

CORPORATE HEADQUARTERS: 1800 GREEN HILLS ROAD, P.O. BOX 660001, SCOTTS VALLEY, CA 95067-0001, (408) 438-5300. OFFICES IN: AUSTRALIA, DENMARK, FRANCE, GERMANY, ITALY, JAPAN, NEW ZEALAND, SINGAPORE, SWEDEN AND THE UNITED KINGDOM • PART #14MN-API01-10 • BOR 2978