Turbo Pascal® for Windows

Windows Reference Guide

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

R1

10 9 8 7 6 5 4 3

C O N T E N T S

T A B L E S

INTRODUCTION

This manual provides comprehensive documentation of all the facilities included with Microsoft Windows, including all the constants, styles, messages, and API functions in the Windows interface, along with full documentation of the objects in the ObjectWindows library.

This volume is divided into two parts. Part 1 covers the Windows API in four chapters. This is the Pascal interface to Windows.

- Chapter 1, "Windows styles and constants," describes all the constants and styles in *WinTypes*, the Turbo Pascal unit that defines the types and constants used by the Windows API.
- Chapter 2, "Windows function reference," is an alphabetical lookup chapter of all the functions and procedures that make up the Windows API, as accessed through *WinProcs*, the Turbo Pascal unit that serves as an import library for the Windows API.
- Chapter 3, "Windows message reference," lists and explains all the messages used by Windows to interact with the applications running in its environment.
- Chapter 4, "Windows type reference," documents all the types defined in the Windows interface, including simple types and record structures.

Part 2 contains the reference material for the ObjectWindows library.

- Chapter 5, "Object Windows reference," lists all the objects in the ObjectWindows library alphabetically, with details about each of the fields and methods for each object type.
- Chapter 6, "Global reference," lists types, constants, and routines defined in ObjectWindows. Essentially, everything in ObjectWindows that is *not* an object in the hierarchy is documented here.

PART 1

Windows API Reference

This part contains essential information about the Microsoft Windows application programming interface (API). The Windows API contains functions, messages, data structures and data types that comprise the framework of the Windows interface. Turbo Pascal developers have full access to Windows functions, although they use them only in cases where ObjectWindows's supplied objects don't suffice.

Definitions for data structures, such as *TRect* and *TWndClass*, appear in the file WINTYPES.INT and in Chapter 4, "Windows type reference." Often, this guide refers to Windows-defined constants, such as *ws_Border* and *sw_ShowWindow*, which are defined in WINTYPES.INT and described in Chapter 1, "Windows styles and constants." You will also use these constants as arguments in ObjectWindows method calls. Chapter 2, "Windows function reference," alphabetically lists all available Windows functions. Chapter 3, "Windows message reference," alphabetically lists all available Windows messages.

Use of Windows API functions, messages, and styles are all explained in detail in the *Windows Programming Guide*, especially in Chapter 7, "Object Windows overview."

C H A P T E R

1

Windows styles and constants

This chapter defines the most-used constants relating to calling Windows functions and receiving Windows messages (see Chapters 2 and 3). The constants are grouped by common use, and these groups are alphabetized.

Where possible, the constants have been grouped and alphabetized by their identifier prefixes (all the constants starting with *color_*, for example, are in one entry, "color_ System color codes." Some of the constants, however, do not have common prefixes, so these have been grouped by their common functionality. For example, the constants *ComplexRegion*, *NullRegion*, and *SimpleRegion* all specify types of graphic regions, and all are used as parameters to the same functions, so they have been grouped together under the heading "Region flags."

Background modes

These constants specify the how the background area is drawn when text or graphics with a hatched brush are drawn. They are used in the *GetBkMode* and *SetBkMode* functions.

Table 1.1
Background modes

Constant	Meaning
Opaque	The background is filled with the current background color.
Transparent	The background is left alone.

bi_ Bitmap compression constants

These constants are used as values in the *biCompression* field of the Windows record type *TBitmapInfoHeader* passed in calls to *CreateDIBitmap*.

Table 1.2 Bitmap compression constants

Constant	Meaning
bi_RGB	The bitmap is not compressed.
bi_RLE8	The bitmap uses a run-length encoded format with 8 bits per pixel.
bi_RLE4	The bitmap uses a run-length encoded format with 4 bits per pixel.

bn_ Button notification codes

These control notification codes are passed in *wm_Command* messages generated by button controls. They indicate the action that has occurred.

Table 1.3 Button notification codes

Constant	Meaning
bn_Clicked	This code notifies the parent of a button that the button has been clicked.
bn_DoubleClicked	This code notifies the parent of a button that the button has been double-clicked. This code only applies to buttons with the *bs_OwnerDraw* style and radio buttons.

bs_ Brush styles

These constants specify logical brush styles and are used in the *lbStyle* field of the *LogBrush* record passed in the *CreateBrushIndirect* function.

Table 1.4 Brush styles

Constant	Meaning
bs_DIBPattern	The brush pattern is specified by a device-independent bitmap.
bs_Hatched	A hatched brush.
bs_Hollow	A hollow, or null, brush.
bs_Null	Same as *bs_Hollow*.
bs_Pattern	The brush pattern is specified by a memory bitmap.
bs_Solid	A solid brush.

bs_ Button styles

These constants are used to specify button styles when creating buttons with the *CreateWindow* and *CreateWindowEx* functions.

Table 1.5
Button styles

Constant	Meaning
bs_3State	This style of button is a box that may be checked, unchecked, and grayed. Graying is typically used to show that the box is disabled.
bs_Auto3State	This style of button is the same as *bs_3State*, except that the box toggles state automatically when clicked.
bs_AutoCheckBox	This style of button is the same as *bs_CheckBox*, except that the box toggles state automatically when clicked.
bs_AutoRadioButton	This style of button is the same as *bs_RadioButton*, except that when the button is clicked, the button is checked and the checkmarks are removed from all other buttons in the same group.
bs_CheckBox	This style of button is a box that may be checked and unchecked. Associated text is placed to the right of the box.
bs_DefPushButton	This style of button is the same as *bs_PushButton*, except that in addition this button is the default selection unless another button or box is selected using the keyboard or mouse.
bs_GroupBox	This style of button is a box for grouping other buttons. Associated text is placed in the upper left corner.
bs_LeftText	Used with *bs_3State*, *bs_CheckBox*, or *bs_RadioButton*, this style causes associated text to be placed to the left of the button or box instead of to the right.
bs_OwnerDraw	This style of button is an owner-draw button. In addition to normal notification codes sent via *wm_Command* messages, the parent also receives requests to paint, invert, and disable the button.
bs_PushButton	This style of button is a button with any associated text placed inside.
bs_RadioButton	This style of button is a small round button that can be checked and unchecked. Associated text is placed to the right of the button. Radio buttons are usually used in groups where one and only one button is checked at a time.

cb_ Combo box return values

These values are returned from Windows combo box messages such as *cb_AddString*. A negative value indicates an error.

Table 1.6 Combo box message return values

Constant	Meaning
cb_Err	An error has occurred and the action was not performed.
cb_ErrSpace	There is not enough space in the combo box to perform the action.
cb_Okay	No error.

cbm_Init CreateDIBitmap constant

When passed in the *Usage* parameter of the function *CreateDIBitmap*, *cbm_Init* indicates that the memory bitmap is to be initialized.

cbn_ Combo box notification codes

These control notification codes are passed in *wm_Command* messages generated by combo box controls. They indicate the action that has occurred.

Table 1.7 Combo box notification codes

Constant	Meaning
cbn_DblClk	This code notifies the parent of a combo box that an entry in the combo box's list box has been double-clicked with the mouse.
cbn_DropDown	This code notifies the parent of a combo box that the combo box's list box is about to be dropped down.
cbn_EditChange	This code notifies the parent of a combo box that the text of the combo box's edit control has changed.
cbn_EditUpdate	This code notifies the parent of a combo box that the combo box's edit control is about to redisplay its text because the text has been modified.
cbn_ErrSpace	This code notifies the parent of a combo box that the system has run out of memory.
cbn_KillFocus	This code notifies the parent of a combo box that the combo box has lost the input focus.
cbn_SelChange	This code notifies the parent of a combo box that the selection in the combo box's list box has been changed.
cbn_SetFocus	This code notifies the parent of a combo box that the combo box has received the input focus.

cbs_ Combo box styles

These constants are used to specify combo box styles when creating combo boxes with the *CreateWindow* and *CreateWindowEx* functions.

Table 1.8
Combo box styles

Constant	Meaning
cbs_AutoHScroll	This style of combo box scrolls the text in the edit control to the right when the user types a character at the end of the line. Without this style, no text is allowed outside the boundary of the edit control.
cbs_DropDown	This style is the same as *cbs_Simple*, except that the list box is displayed only when an icon next to the selection field is selected.
cbs_DropDownList	This style is the same as *cbs_DropDown*, except that a static text item is used to display the current selection instead of an edit control.
cbs_HasStrings	This style may be used in conjunction with the *cbs_OwnerDrawFixed* or *cbs_OwnerDrawVariable* style. This style of combo box uses strings as its entries. The strings are managed by the system and may be retrieved using the *cb_GetLBText* message.
cbs_NoIntegralHeight	This style of combo box is exactly the size given when the combo box is created. Normally the size of the combo box used to create a combo box may be changed so that the combo box does not display partial entries.
cbs_OEMConvert	This style may be used in conjunction with the *cbs_Simple* or *cbs_DropDown* styles. This style of combo box translates each character entered into the combo box's edit control from the ANSI character set to the OEM character set and then back to ANSI. The *AnsiToOem* function will then behave correctly when it is applied to entries in the combo box's list box or to the text in the combo box's edit control. The *cbs_OEMConvert* style is useful for combo boxes which contain file names.
cbs_OwnerDrawFixed	This style of combo box must be drawn by its owner. The entries in the combo box's list box are all the same height.
cbs_OwnerDrawVariable	This style of comb-box must be drawn by its owner. The entries in the combo box's list box are variable in height.
cbs_Simple	This style of combo box has its list box displayed at all times. The current list box selection is displayed in the edit control.

Table 1.8: Combo box styles (continued)

cbs_Sort	This style of combo box has a sorted list box. The sort order may be different for combo boxes with the *cbs_OwnerDrawFixed* or *cbs_OwnerDrawVariable* style.

cc_ Curve capabilities

These constants represent the curve rendering capabilities of a device.

Table 1.9
Curve capability constants

Constant	Meaning
cc_None	Curves not supported.
cc_Chord	Can do chord arcs.
cc_Circles	Can do circles.
cc_Ellipses	Can do ellipses.
cc_Interiors	Can do interiors.
cc_Pie	Can do pie wedges.
cc_Styled	Can do styled lines.
cc_Wide	Can do wide lines.
cc_WideStyled	Can do wide styled lines.

ce_ Comm error flags

These flags specify a communications error when returned from the *GetCommError* function.

Table 1.10
Communication error flags

Constant	Meaning
ce_Break	Break detected.
ce_CTSTO	Clear-to-send timeout.
ce_DNS	LPTx device is not selected.
ce_DSRTO	Data-set-ready timeout.
ce_Frame	Framing error.
ce_IOE	LPTx I/O error.
ce_Mode	Requested mode unsupported or invalid Cid.
ce_OOP	LPTx out-of-paper.
ce_Overrun	Received character is overrun by next character and not read.
ce_PTO	LPTx timeout.
ce_RLSDTO	Receive-line-signal-detect timeout.
ce_RxOver	Receive queue overflow.
ce_RxParity	Parity error.
ce_TxFull	Transmit queue is full.

cf_ Clipboard formats

These constants identify clipboard formats when passing data to and from the Windows clipboard. They are also used in the dynamic data exchange (DDE) protocol. They are use in the following functions: *EnumClipboardFormats*, *GetClipboardData*, *GetClipboardFormatName*, *GetPriorityClipboardFormat*, *IsClipboardFormatAvailable*, and *SetClipboardData*.

Table 1.11
Clipboard formats

Constant	Meaning
cf_Bitmap	Windows bitmap.
cf_DIB	Windows device independent bitmap.
cf_DIF	Software Arts' Data Interchange Format.
cf_DSPBitmap	A private bitmap format's display information.
cf_DSPMetaFilePict	A private metafile format's display information.
cf_DSPText	A private text format's display information.
cf_MetaFilePict	Windows metafile.
cf_OEMText	Text using the OEM character set.
cf_OwnerDisplay	Owner display format.
cf_Palette	A color palette if needed for specific color rendering clipboard data.
cf_PrivateFirst	The first in a range of values for private data formats. Handles to data in these formats will not be automatically freed and should freed by the application before closing.
cf_PrivateLast	The last in the range of values for private data formats.
cf_SYLK	Microsoft Symbolic Link format.
cf_Text	Text.
cf_TIFF	Tag Image File Format.

cchDeviceName Device name length constant

The *cchDeviceName* constant limits the size of a device name string to 32 characters.

clip_ Font clipping precision flags

These constants specify the clipping precision of fonts produced with the *CreateFont* function. They govern how text appears when it is partially outside the clipping region.

clip_Character_Precis

clip_Default_Precis
clip_Stroke_Precis

color_ System color codes

These codes specify aspects of the Windows user interface for which the programmer can directly set the color with the *SetSysColors* function.

Table 1.12
System color codes

Constant	Meaning
color_ActiveBorder	Active window border.
color_ActiveCaption	Turbo Pascal window caption.
color_AppWorkSpace	Background of MDI frame windows.
color_Background	Windows desktop.
color_BtnFace	The face of push buttons.
color_BtnShadow	The edge of push buttons.
color_BtnText	The text of push buttons.
color_CaptionText	Caption text, size box and scroll bar arrow box.
color_GrayText	Grayed text.
color_Highlight	A control's selected items.
color_HighlightText	The text of a control's selected items.
color_InactiveBorder	Inactive window border.
color_InactiveCaption	Inactive window caption.
color_Menu	Menu bar background.
color_MenuText	Menu bar text.
color_ScrollBar	Scroll bar background.
color_Window	Window background.
color_WindowFrame	Window frame.
color_WindowText	Text in a window.

com_ Communication device status flags

These flags can be used as bit masks to retrieve or set bit fields that make up the *Flags* field of a *TComStat* record.

Table 1.13
TComStat bit flag constants

Constant	Bit accessed
com_CtsHold	*fCtsHold*
com_DsrHold	*fDsrHold*
com_RlsdHold	*fRlsdHold*
com_XoffHold	*fXoffHold*
com_XoffSent	*fXoffSent*
com_Eof	*fEof*
com_Txim	*fTxim*

C

Comm configuration constants

One of the following constants are used in the *Parity* field of the *TDCB* record type. They indicate the type of parity to be used by a serial communications device.

Table 1.14 Communication parity constants

Constant	Parity setting
EvenParity	Even
MarkParity	Mark
NoParity	No parity
OddParity	Odd
SpaceParity	Space

One of the following constants is used in the *StopBits* field of the *TDCB* record type. They indicate the number of stop bits to be used by a serial communications device.

Table 1.15 Communication stop bits constants

Constant	Stop bits used
OneStopBit	One
One5StopBits	1.5
TwoStopBits	Two

cp_ Clipping capabilities

These constants indicate the clipping abilities of the device.

Table 1.16 Clipping capability constants

Constant	Meaning
cp_None	No clipping of output.
cp_Rectangle	Output clipped to rectangles.

cs_ Class styles

These window class style constants are used in the style field of the *WNDCLASS* data structure. They can be combined using or.

Table 1.17
Class styles

Constant	Meaning
cs_ByteAlignClient	The window's client area is aligned on the byte boundary, in the x direction.
cs_ByteAlignWindow	The window is aligned on the byte boundary, in the x direction.
cs_ClassDC	Instances of the window class share their own display context.
cs_DblClks	The window will get mouse double-click messages.
cs_GlobalClass	The window class can be used by all running applications.
cs_HRedraw	The entire window will be redrawn if its horizontal size changes.
cs_NoClose	The window's Control menu's Close option is disabled.
cs_OwnDC	Each window instance gets its own display context. Uses 800 bytes of memory per window.
cs_ParentDC	The window uses its parent's display context.
cs_SaveBits	The window's contents are saved to a bitmap when they are not currently displayed. This bitmap is used to redisplay its contents. Use minimally.
cs_VRedraw	The entire window will be redrawn if its vertical size changes.

ctlcolor_ Control color flags

These flags indicate the type of control or dialog box that is about to be drawn, and is passed in a *wm_CtlColor* message.

Table 1.18
Control color flags

Constant	Meaning
ctlcolor_Btn	A button control
ctlcolor_Dlg	A dialog box
ctlcolor_Edit	An edit control
ctlcolor_ListBox	A list box
ctlcolor_Max	A maximum control
ctlcolor_MsgBox	A message box
ctlcolor_Scrollbar	A scroll bar
ctlcolor_Static	A static control

cw_UseDefault constant

cw_UseDefault is used as a parameter to *CreateWindow* or *CreateWindowEx*. It tells Windows to assign a default size or position to the window being created.

dc_ Device capabilities index constants

These constants are used in the *Index* parameter of calls to *DeviceCapabilities* to indicate the specific capability of the printer driver to query. The *Index* argument can be any one of these values:

Table 1.19 Device capabilities index constants

Constant	Capability requested
dc_BinNames	Information on available paper bins
dc_Bins	The list of available bins
dc_Driver	The printer driver version number
dc_Duplex	The level of duplex printing support
dc_Extra	The number of bytes at the end of the *TDevMode* record to be used for device-specific data
dc_Fields	The dmFields field of the *TDevMode* record
dc_MaxExtent	The maximum paper size
dc_MinExtent	The minimum paper size
dc_Papers	A list of supported paper sizes
dc_Papersize	A list of the exact sizes of the supported paper sizes
dc_Size	The *dmSize* field of the *TDevMode* record
dc_Version	The specification version the printer driver uses

dcb_ Communication device control block flags

These flags can be used as bit masks to retrieve or set bit fields that make up the *Flags* field of a *TDCB* record.

Table 1.20 *TDCB* record communication device control block flags

Constant	Bit accessed
dcb_Binary	*fBinary*
dcb_RtsDisable	*fRtsDisable*
dcb_Parity	*fParity*
dcb_OutxCtsFlow	*fOutxCtsFlow*
dcb_OutxDsrFlow	*fOutxDsrFlow*
dcb_DtrDisable	*fDtrDisable*
dcb_OutX	*fOutX*
dcb_InX	*fInX*
dcb_PeChar	*fPeChar*

Table 1.20: *TDCB* record communication device control block flags (continued)

dcb_Null	*fNull*
dcb_ChEvt	*fChEvt*
dcb_Dtrflow	*fDtrflow*
dcb_Rtsflow	*fRtsflow*

dde_ DDE record type flags

These flags can be used as bit masks to retrieve or set bit fields that make up the *Flags* field of a *TDDEAdvise*, *TDDEData*, or *TDDEPoke* record.

Table 1.21
DDE record type flags

Mask	Bit field	Used in record type
dde_AckReq	*fAckReq*	*TDDEAdvise, TDDEData*
dde_DeferUpdt	*fDeferUpd*	*TDDEAdvise*
dde_Response	*fResponse*	*TDDEData*
dde_Release	*fRelease*	*TDDEData, TDDEPoke*

dde_ DDE Return codes

These bitmasks are used to specify bits in the *TDDEAck* record:

Table 1.22
DDE return codes

dde_Ack	The *fAck* bit of the *TDDEAck.Status* word. If *LParamLo* and *dde_Ack* = 1, the request is accepted. If *LParamLo* and *dde_Ack* = 0, the request is denied.
dde_AppReturnCode	Reserved for application-specific return codes.
dde_Busy	The *fBusy* bit of the *TDDEAck.Status* word. If the request is denied, *LParamLo* and *dde_Busy* = 1 indicated that the application was unable to respond.

Device capabilities

These constants are associated with particular capabilities of a device context. They are used in the *GetDeviceCaps* function.

Table 1.23
Device capability constants

Constant	Meaning
AspectX	Relative pixel width.
AspectXY	Diagonal pixel length.
AspectY	Relative pixel height.
BitsPixel	Number of bits per pixel.
ClipCaps	Clipping capabilities. Returns a cp_ clipping capabilities constant. See "(cp_) Clipping capabilities" in this section.
ColorRes	Actual color resolution in bits per pixel.
CurveCaps	Curve capabilities. Returns a cc_ curve capabilities constant. See "(cc_) Curve capabilities" in this section.
DriverVersion	Device driver version. For example $100 is 1.0.
HorzRes	Horizontal width in pixels.
HorzSize	Horizontal size in millimeters.
LineCaps	Line capabilities. Returns a lc_ line capabilities constant. See "(lc_ Line capabilities" in this section.
LogPixelsX	Number of pixels per horizontal inch.
LogPixelsY	Number of pixels per vertical inch.
NumBrushes	Number of brushes the device has.
NumColors	Number of colors the device supports.
NumFonts	Number of fonts the device has.
NumMarkers	Number of markers the device has.
NumPens	Number of pens the device has.
NumReserved	Number of reserved entries in palette.
PDeviceSize	Size required for device descriptor.
Planes	Number of color planes.
PolygonalCaps	Polygonal capabilities. Returns a pc_ polygonal capabilities constant. See "(pc_) Polygonal capabilities" in this section.
RasterCaps	Raster capabilities. Returns a rc_ raster capabilities constant. See "(rc_) Raster capabilities" in this section.
SizePalette	Number of entries in physical palette.
Technology	Device classification. Returns a dt_ device technology constant. See "(dt_) Device technologies" in this section.
TextCaps	Text capabilities. Returns a tc_ text capabilities constant. See "(tc_) Text capabilities" in this section.
VertRes	Vertical width in pixels.
VertSize	Vertical size in millimeters.

DIB_ Color table identifiers

These constants specify the how colors are accessed for use in device-independent bitmaps. They are used in *CreateDIBitmap*, *CreateDIBPatternBrush*, *GetDIBits*, *SetDIBits*, *SetDIBitsToDevice*, and *StretchDIBits* functions.

Table 1.24
Color table identifiers

Constant	Meaning
DIB_Pal_Colors	The color table is an array of 16-bit indexes into the currently realized logical palette.
DIB_RGB_Colors	The color table holds RGB literal values.

dlgc_ Dialog codes

These codes specify types of dialog input the application, rather than Windows, will process. They are return values from the *wm_GetDlgCode* message.

Table 1.25
Dialog codes

Constant	Meaning
dlgc_DefPushButton	Default push button messages.
dlgc_HasSetSel	*em_SetSel* messages.
dlgc_RadioButton	Radio button messages.
dlgc_UndefPushButton	Push button messages.
dlgc_WantAllKeys	All keyboard input messages
dlgc_WantArrows	Arrow key messages.
dlgc_WantChars	*wm_Char* messages.
dlgc_WantMessage	All keyboard input messages. The messages will be passed to the control.
dlgc_WantTab	*Tab* key messages.

DlgWindowExtra Dialog class constant

When defining a new class for a dialog from a resource, set the *cbWndExtra* field of the *TWndClass* record to *DlgWindowExtra*.

dm_ TDevMode field selection constants

These constants are used as field selection bits. Each constant represents a specific bit in the *dmFields* field of a *TDevMode* record. If the specified bit is set, it indicates that the corresponding field in the record has been initialized.

Table 1.26 Device mode field selection constants

Constant	Field
dm_Color	*dmColor*
dm_Copies	*dmCopies*
dm_DefaultSource	*dmDefaultSource*
dm_Duplex	*dmDuplex*
dm_Orientation	*dmOrientation*
dm_PaperLength	*dmPaperLength*
dm_PaperSize	*dmPaperSize*
dm_PaperWidth	*dmPaperWidth*
dm_PrintQuality	*dmPrintQuality*
dm_Scale	*dmScale*

dm_ Device mode selections

These constants are used in the *Mode* parameter of the *ExtDeviceMode* function to specify operations. *Mode* may be one or more of the following:

Table 1.27 ExtDeviceMode mode constants

Constant	Meaning
dm_Copy	Writes the printer driver's settings to the *TDevMode* record passed in *DevModeOutput*.
dm_Modify	Modifies the printer driver's settings to be what's found in the *TDevMode* record passed in *DevModeInput*.
dm_Prompt	Produces the printer driver's setup dialog box and sets the printer to the settings selected by the user.
dm_Update	Updates the printer environment and the WIN.INI file with the printer's current settings.

dmbin_ Device mode bin selection constants

These constants are used in the *dmDefaultSource* field of a *TDevMode* record, indicating from which bin paper will feed by default.

dmbin_Auto
dmbin_Cassette
dmbin_Envelope
dmbin_EnvManual
dmbin_LargeCapacity
dmbin_LargeFmt
dmbin_Lower
dmbin_Manual
dmbin_Middle
dmbin_OnlyOne
dmbin_SmallFmt
dmbin_Tractor
dmbin_Upper

dmcolor_ Device mode color constants

These constants are used in the *dmColor* field of a *TDevMode* record, indicating whether the printer supports color printing.

Table 1.28
Device mode color constants

Constant	Meaning
dmcolor_Monochrome	Printer is black-and-white only
dmcolor_Color	Printer prints in color

dmdup_ Device mode duplex constants

These constants are used in the *dmDuplex* field of a *TDevMode* record to indicate the use of duplex (double-sided) printing.

Table 1.29
Printer duplex selection constants

Constant	Meaning
dmdup_Horizontal *dmdup_Simplex* *dmdup_Vertical*	Print on only one side

dmorient_ Device mode orientation constants

These constants are used in the *dmOrientation* field of a *TDevMode* record to indicate the desired printing orientation.

Table 1.30 Printer orientation constants

Constant	Meaning
dmorient_Portrait	Use portrait orientation
dmorient_Landscape	Use landscape orientation

dmpaper_ Device mode paper type constants

These constants are used in the *dmPaperSize* field of a *TDevMode* record, indicating the size of the paper being printed on.

Table 1.31 Printer paper size constants

Constant	Width	Height	Units
dmpaper_10X14	10	14	inches
dmpaper_11X17	11	17	inches
dmpaper_A3	297	420	mm
dmpaper_A4	210	297	mm
dmpaper_A4Small	210	297	mm
dmpaper_A5	148	210	mm
dmpaper_B4	250	354	mm
dmpaper_B5	182	257	mm
dmpaper_CSheet	C size sheet		
dmpaper_DSheet	D size sheet		
dmpaper_Env_10	4 1/8	9 1/2	inches
dmpaper_Env_11	4 1/2	10 3/8	inches
dmpaper_Env_12	4 3/4	11	inches
dmpaper_Env_14	5	11 1/2	inches
dmpaper_Env_9	3 7/8	8 7/8	inches
dmpaper_ESheet	E size sheet		
dmpaper_Executive	7 1/2	10	inches
dmpaper_Folio	8 1/2	13	inches
dmpaper_Ledger	17	11	inches
dmpaper_Legal	8 1/2	14	inches
dmpaper_Letter	8 1/2	11	inches
dmpaper_LetterSmall	8 1/2	11	inches
dmpaper_Note	8 1/2	11	inches
dmpaper_Quarto	215	275	mm
dmpaper_Statement	5 1/2	8 1/2	inches
dmpaper_Tabloid	11	17	inches

dmres_ Device mode resolution constants

These constants are used in the *dmPrintQuality* field of a *TDevMode* record to indicate the resolution of the printer being used.

Table 1.32 Printer resolution constants

Constant	Meaning
dmres_Draft	Draft quality
dmres_Low	Low print quality
dmres_Medium	Medium print quality
dmres_High	High print quality

drive_ Drive types

These constants specify the disk drive type and are returned from the *GetDriveType* function.

Table 1.33 Drive type constants

Constant	Meaning
drive_Fixed	The disk cannot be removed from the drive.
drive_Remote	The disk is at a remote (networked) location.
drive_Removeable	The disk can be removed from the drive.

ds_ Dialog styles

These constants are used to specify dialog styles when creating dialog boxes with the *CreateWindow* and *CreateWindowEx* functions.

Table 1.34 Dialog styles

Constant	Meaning
ds_AbsAlign	This style of dialog box is aligned relative to the upper-left corner of the screen instead of relative to the upper-left corner of the owner of the dialog box.
ds_LocalEdit	This style of dialog box uses memory in the application's data segment for data structures associated with controls which are children of the dialog box. The default is to use global memory. If the *ds_LocalEdit* style is not used, the *em_GetHandle* and *em_SetHandle* messages must not be used.
ds_ModalFrame	This style of dialog box has a modal dialog box frame. The *ws_Caption* and *ws_SysMenu* styles may be used with the *ds_ModalFrame* style.
ds_NoIdleMsg	This style of dialog box excludes *wm_EnterIdle* messages that are normally sent to the owner of the dialog box.

Table 1.34: Dialog styles (continued)

ds_SetFont	A dialog with this style sets its own font for drawing its controls. An extended form of the DLGTEMPLATE structure must be used to create the dialog. It includes the basic DLGTEMPLATE structure header followed immediately by a font-information structure. A dialog with the *ds_SetFont* style is sent a *wm_SetFont* message before the dialog's controls are created.
ds_SysModal	This style of dialog box suspends all Windows applications while it is displayed.

dt_ Device technologies

These constants represent the type of device being used.

Table 1.35 Device type constants

Constant	Meaning
dt_CharStream	Character stream.
dt_DispFile	Display file.
dt_MetaFile	Metafile.
dt_Plotter	Vector plotter.
dt_RasCamera	Raster camera.
dt_RasDisplay	Raster display.
dt_RasPrinter	Raster printer.

dt_ Text drawing formatting flags

These constants specify formatting options for the *DrawText* function.

Table 1.36 *DrawText* formatting options

Constant	Meaning
dt_Bottom	Bottom-justified.
dt_CalcRect	Recalculate the bounding rectangle to fit the text, but do not actually draw the text.
dt_Center	Centered horizontally.
dt_ExpandTabs	Expands tabs.
dt_ExternalLeading	Includes a font's external leading in text height.
dt_Internal	Includes a font's internal leading in text height.
dt_Left	Left-justified.
dt_NoClip	Draw without clipping.
dt_NoPrefix	Do not process prefix characters, such as '&'.
dt_Right	Right-justified.
dt_SingleLine	Single line only.
dt_TabStop	Sets tab stops.
dt_Top	Top-justified.

Table 1.36: *DrawText* formatting options (continued)

dt_VCenter	Centered vertically.
dt_WordBreak	Word wrap.

en_ Edit control notification codes

These control notification codes are passed in *wm_Command* messages generated by edit controls. They indicate the action that has occurred.

Table 1.37 Edit control notification codes

Constant	Meaning
en_Change	This code notifies the owner of an edit control that the edit control's text has changed.
en_ErrSpace	This code notifies the owner of an edit control that the system has run out of memory.
en_HScroll	This code notifies the parent of an edit control that the edit control's horizontal scroll bar has been clicked. The parent is notified before the screen is updated.
en_KillFocus	This code notifies the parent of an edit control that the edit control has lost the input focus.
en_MaxText	This code notifies the parent of an edit control that the maximum number of characters allowed in the edit control was exceeded by the last insertion.
en_SetFocus	This code notifies the parent of an edit control that the edit control has received the input focus.
en_Update	This code notifies the parent of an edit control that the edit control is about to redisplay its text because the text has been modified.
en_VScroll	This code notifies the parent of an edit control that the edit control's vertical scroll bar has been clicked. The parent is notified before the screen is updated.

es_ Edit control styles

These constants are used to specify edit control styles when creating edit controls with the *CreateWindow* and *CreateWindowEx* functions.

Table 1.38
Edit control styles

Constant	Meaning
es_AutoHScroll	This style of edit control automatically scrolls its text to the right by 10 characters when a character is typed at the end of the line. The text is scrolled back to position zero when the *Enter* key is pressed.
es_AutoVScroll	This style of edit control automatically scrolls its text up one page when the ENTER key is pressed with the caret on the last line.
es_Center	This style of edit control centers its text. The *es_Center* style may be used only if the *es_MultiLine* style is used also.
es_Left	This style of edit control aligns its text flush-left. The *es_Left* style may be used only if the *es_MultiLine* style is used also.
es_LowerCase	This style of edit control converts all characters to lowercase as they are typed.
es_MultiLine	This style of edit control is a multiple-line edit control. The *es_AutoVScroll* style may only be used with the *es_MultiLine* style. If the *es_AutoVScroll* style is not used, a beep is sounded when the *Enter* key is pressed with the caret on the last line. If the *es_AutoHScroll* style is not used, new words typed in are automatically wrapped to the next line when necessary. The wordwrap positions will change when the window is re-sized. A multiple-line edit control with scroll bars handles its own scroll bar messages, otherwise scrolling is done automatically as described above.
es_NoHideSel	This style of edit control does not hide its selection when the edit control loses the input focus. The default behavior is to hide the selection when the edit control loses the focus.
es_OEMConvert	This style of edit control converts entered text from the ANSI character set to the OEM character set and then back to ANSI. The AnsiToOem function will then behave correctly when it is applied to the edit control's text. The *es_OEMConvert* style is useful for edit controls which contain file names.
es_Password	All characters typed into the edit control are displayed as an asterisk (*). The *em_SetPasswordChar* message can be used to change the displayed character.
es_Right	This style of edit control aligns its text flush-right. The *es_Right* style may be used only if the *es_MultiLine* style is used also.
es_UpperCase	This style of edit control converts all characters to uppercase as they are typed.

Escape comm constants

These constants specify a function code to be carried out by a communication device. Use them in calls to the *EscapeCommFunction* function.

Table 1.39 *EscapCommFunction* constants

Constant	Meaning
ClrDTR	Clears the data-set-ready signal.
ClrRTS	Clears the request-to-send signal.
ResetDev	Resets the device if possible.
SetDTR	Clears the data-terminal-ready signal.
SetRTS	Clears the request-to-send signal.
SetXoff	Simulates receiving an XOFF character.
SetXon	Simulates receiving an XON character.

eto_ ExtTextOut options

These options specify the background painting method used by the *ExtTextOut* function.

Table 1.40 *ExtTextOut* options

Constant	Meaning
eto_Clipped	The text is clipped to the bounding rectangle.
eto_Opaque	The background color fills the bounding rectangle.

ev_ Comm event constants

These constants serve as pointers into a communication device event mask and are return values in the *SetCommEventMask* function. The following listing describes the state under which the mask bits pointed to are set (equal to 1).

Table 1.41 *SetCommEventMask* return values

Constant	Meaning
ev_Break	Break received.
ev_CTS	Clear-to-send state changes.
ev_DSR	Data-set-ready state changes.
ev_Err	Line status error occurred.
ev_PErr	Printer error occurred.
ev_Ring	A ring indicator is detected.
ev_RLSD	Receive-line-signal-detect state changes.
ev_RxChar	Any character is received.

Table 1.41: *SetCommEventMask* return values (continued)

ev_RxFlag	The event character specified in the device's control block is received.
ev_TxEmpty	The transmit queue is empty.

Flood fill style flags

These flags specify the type of flood fill used in the *ExtFloodFill* function.

Table 1.42
Flood fill style flags

Constant	Meaning
FloodFillBorder	The area up to the specified color will be filled. This is the type of flood fill performed by the *FloodFill* function.
FloodFillSurface	The area containing the specified color will be filled. This type of flood fill is most appropriate for complex, multi-colored surfaces.

ff_ Font family flags

These constants specify the desired character set for fonts produced with the *CreateFont* function.

Table 1.43
CreateFont character set options

Constant	Meaning
ff_Decorative	Novelty fonts, such as Old English.
ff_DontCare	Don't care or don't know.
ff_Modern	Constant stroke width, serifed or sans-serifed. For example Pica, Elite, and Courier.
ff_Roman	Variable stroke width, serifed. For example, Times Roman and Century Schoolbook.
ff_Script	Hand written fonts, such as Script.
ff_Swiss	Variable stroke width, sans-serifed. For example, Helvetica and Swiss.

Font character set flags

These constants specify the desired character set for fonts produced with the *CreateFont* function.

ANSI_CharSet
Symbol_CharSet
ShiftJis_CharSet
OEM_CharSet

Font output quality flags

These constants specify the desired output quality of fonts produced with the *CreateFont* function.

Table 1.44 *CreateFont* output quality options

Constant	Meaning
Default_Quality	Appearance is not important.
Draft_Quality	Appearance is only moderately important.
Proof_Quality	Character quality is paramount. It is more important than matching the logical font attributes.

Font pitch flags

These constants specify the desired pitch of fonts produced with the *CreateFont* function.

Default_Pitch
Fixed_Pitch
Variable_Pitch

fw_ Font weight flags

These constants specify the desired font weight for fonts produced with the *CreateFont* function. The list is presented in order of lightest (*fw_Thin*) to heaviest (*fw_Heavy*).

fw_DontCare
fw_Thin
fw_ExtraLight, fw_UltraLight
fw_Light
fw_Normal, fw_Regular
fw_Medium
fw_SemiBold, fw_DemiBold
fw_Bold, fw_Black
fw_ExtraBold, fw_UltraBold
fw_Heavy

gcl_ Class field offsets

These constants specify the byte offset of the window class information to be retrieved or modified with the *GetClassLong* or *SetClassLong* functions.

Table 1.45
Class field offsets for *GetClassLong* or *SetClassLong*

Constant	Meaning
gcl_MenuName	The pointer to the menu name (*SetClassLong* only).
gcl_WndProc	The pointer to the window function.

gcw_ Class field offsets

These constants specify the byte offset of the window class information to be retrieved or modified with the *GetClassWord* or *SetClassWord* functions.

Table 1.46
Class field offsets for *GetClassWord* or *SetClassWord*

Constant	Meaning
gcw_CBClsExtra	Retrieves or sets to two the number of extra bytes of class information.
gcw_CBWndExtra	Retrieves or sets to two the number of extra bytes of window information.
gcw_HBrBackground	The handle to the background brush.
gcw_HCursor	The handle to the cursor.
gcw_HIcon	The handle to the icon.
gcw_HModule	The handle to the module. (*GetClassWord* only).
gcw_Style	The window class style bits.

gmem_ Global memory flags

These flags specify characteristics of the global memory block created with the functions *GlobalAlloc* and *GlobalReAlloc*. They are also used in the *GlobalFlags* and *GetFreeSpace* functions.

Table 1.47 Global memory flags

Constant	Meaning
gmem_DDEShare	The memory block can be shared by many applications using the dynamic data exchange (DDE) protocol. The block is discarded when the allocating application terminates.
gmem_Discardable	The memory block can be discarded. Must be used with *gmem_Moveable*.
gmem_Discarded	The memory block has been discarded. (*GlobalFlags* only.)
gmem_Fixed	The memory block is fixed in one memory location.
gmem_LockCount	When combined with the low-order byte of the return value from *GlobalFlags*, returns the memory block's reference count. (*GlobalFlags* only.)
gmem_Lower	Same as *gmem_Not_Banked*.
gmem_Modify	The global memory flags are to be changed when this flag is included (*GlobalReAlloc* only).
gmem_Moveable	The memory block is fixed in one memory location.
gmem_NoCompact	When allocating the memory block, no other memory blocks will be compacted or discarded.
gmem_NoDiscard	When allocating the memory block, no other memory blocks will be discarded.
gmem_Not_Banked	The memory block is allocated in non-banked memory.
gmem_Notify	The notification routine will be called if the memory is discarded.
gmem_Share	Same as *gmem_DDEShare*.
gmem_ZeroInit	Initializes the contents of the memory block to zero.

gw_ Get window constants

These constants specify the relationship between the desired window and the window provided in calls to the *GetNextWindow* and *GetWindow* functions. *GetNextWindow* accepts only *gw_HWndNext* and *gw_HWndPrev*.

Table 1.48 Get window constants

Constant	Meaning
gw_Child	The first child window.
gw_HWndFirst	For a child window, its first sibling window.
gw_HWndLast	For a child window, its last sibling window.
gw_HWndNext	The next window on the window manager's list.
gw_HWndPrev	The previous window on the window manager's list.
gw_Owner	The parent window.

gwl_ Window field offsets

These constants specify the byte offset of the window attribute to be retrieved or modified with *GetWindowLong* or *SetWindowLong*.

Table 1.49
Window field offsets for *GetWindowLong* or *SetWindowLong*

Constant	Meaning
gwl_ExStyle	The extended window style.
gwl_Style	The window style.
gwl_WndProc	The pointer to the window function.

gww_ Window field offsets

These constants specify the byte offset of the window information to be retrieved or modified with *GetWindowWord* or *SetWindowWord*.

Table 1.50
Window field offsets for *GetWindowWord* or *SetWindowWord*

Constant	Meaning
gww_HInstance	The instance identifier of the module that owns the window.
gww_HWndParent	The parent window. (*GetWindowWord* only).
gww_ID	The control ID of the child window.

help_ Help commands

These constants indicate to the Windows help facility the type of help requested by the application. Use them in calls to the *WinHelp* function.

Table 1.51
Help command constants

Constant	Meaning
help_Context	Help on a context specified in the Data parameter
help_HelpOnHelp	Help on using the help facility
help_Index	Display the help index
help_Key	Help on a keyword specified in the Data parameter
help_MultiKey	Help on a keyword in an alternate keyword table
help_Quit	Terminate the help facility
help_SetIndex	Display the help index specified in the Data parameter in the file specified in the HelpFile parameter

hs_ Hatch styles

The following constants specify hatch styles for brush tools. They are used in the *CreateHatchBrush* function.

Table 1.52
Hatch style constants

Constant	Meaning
hs_BDiagonal	\\\\\
hs_Cross	+++++
hs_DiagCross	xxxxx
hs_FDiagonal	/////
hs_Horizontal	———
hs_Vertical	|||||

ht Hit test codes

These codes indicate the position of the cursor in relation to a window and are passed in many messages involving placement or movement of the cursor, including *wm_MouseActivate*, *wm_SetCursor*, and *wm_NCHitTest*.

Table 1.53
Hit test codes

Constant	Meaning
htBottom	The window's bottom border
htBottomLeft	The window's bottom left corner
htBottomRight	The window's bottom right corner
htCaption	The caption area
htClient	The client area
htError	Same as *htNowhere*, plus produces a beep
htGrowBox	The size box
htHScroll	The horizontal scroll bar
htLeft	The window's left border
htMenu	The menu area
htNowhere	The screen background or a dividing line between windows
htReduce	The minimize box
htRight	The window's right border
htSize	The same as *htGrowBox*
htSysMenu	The Control-menu box
htTop	The window's top border
htTopLeft	The window's top left border
htTopRight	The window's top right border
htTransparent	A window currently covered by another
htVScroll	The vertical scroll bar
htZoom	The maximize box

id_ Dialog box command IDs

The following constants are return values for the *MessageBox* function. They indicate the result of the message box.

Table 1.54 Dialog box command ID constants

Constant	Meaning
id_Abort	Abort button was pressed.
id_Cancel	Cancel button was pressed.
id_Ignore	Ignore button was pressed.
id_No	No button was pressed.
id_Ok	OK button was pressed.
id_Retry	Retry button was pressed.
id_Yes	Yes button was pressed.

idc_ Standard cursor IDs

These IDs identify one of the stock cursors Windows defines. Load them with the *LoadCursor* function.

Table 1.55 Standard cusor ID constants

Constant	Meaning
idc_Arrow	Arrow.
idc_Cross	Crosshair.
idc_IBeam	I-beam for text.
idc_Icon	Empty icon (concentric squares).
idc_Size	Four-pointed arrow (north, south, east and west).
idc_SizeNESW	Two-pointed arrow (northeast and southwest).
idc_SizeNS	Two-pointed arrow (north and south).
idc_SizeNWSE	Two-pointed arrow (northwest and southeast).
idc_SizeWE	Two-pointed arrow (east and west).
idc_UpArrow	Vertical arrow pointing up.
idc_Wait	Hourglass.

idi_ Standard icon IDs

These IDs identify one of the stock icons Windows defines. Load them with the *LoadIcon* function.

Table 1.56 Standard icon ID constants

Constant	Meaning
idi_Application	Default.
idi_Asterisk	'i' for information messages.
idi_Exclamation	'!' for warnings.
idi_Hand	Stop sign for serious warnings.
idi_Question	'?' for prompting messages.

ie_ Open comm error flags

These flags are negative values that specify a communication device opening error when returned by the *OpenComm* function.

Table 1.57 *OpenComm* error flags

Constant	Meaning
ie_BadID	Invalid ID.
ie_BaudRate	Unsupported baud rate.
ie_ByteSize	Invalid byte size.
ie_Default	Error in default parameters.
ie_Hardware	Hardware is absent.
ie_Memory	Unable to allocate queues.
ie_NOpen	Device not open.
ie_Open	Device already open.

lb_ List box return values

These values are returned from Windows list box messages such as *lb_AddString*. A negative value indicates an error.

Table 1.58 List box message return values

Constant	Meaning
lb_Err	An error has occurred; the action was not performed
lb_ErrSpace	Not enough space in the list box to perform the action
lb_Okay	No error

lbn_ List box notification codes

These control notification codes are passed in *wm_Command* messages generated by list box controls. They indicate the action that has occurred.

Table 1.59
List box notification codes

Constant	Meaning
lbn_DblClk	This code notifies the parent of a list box that an entry has been double-clicked with the mouse.
lbn_ErrSpace	This code notifies the parent of a list box that the system has run out of memory.
lbn_KillFocus	This code notifies the parent of a list box that the list box has lost the input focus.
lbn_SelChange	This code notifies the parent of a list box that the selection in the list box has been changed.
lbn_SetFocus	This code notifies the parent of a list box that the list box has received the input focus.

lbs_ List box styles

These constants are used to specify list box styles when creating list boxes with the *CreateWindow* and *CreateWindowEx* functions.

Table 1.60
List box styles

Constant	Meaning
lbs_ExtendedSel	This style of list box allows multiple items to be selected using the *Shift* key and the mouse or some other special key combination.
lbs_HasStrings	This style may be used with the *lbs_OwnerDrawFixed* or *lbs_OwnerDrawVariable* style. This style of list box uses strings as its entries. The strings are managed by the system and may be retrieved using the *lb_GetText* message.
lbs_MultiColumn	This style of list box has multiple columns which can be scrolled horizontally. The column width can be set using the *lb_SetColumnWidth* message.
lbs_MultipleSel	This style of list allows multiple items to be selected using the mouse. Each time an entry is clicked or double-clicked its selection state is toggled.
lbs_NoIntegralHeight	This style of list box is exactly the size given when the list box is created. Normally the size of the list box used to create a list box may be changed so that the list box does not display partial entries.

Table 1.60: List box styles (continued)

lbs_NoRedraw	This style of list box is not redrawn when changes are made. The *wm_SetRedraw* message is used to set or reset this style dynamically.
lbs_Notify	This style of list box has an input message sent to its parent window whenever an entry is clicked or double-clicked.
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_OwnerDrawFixed	This style of list box must be drawn by its owner. All entries are the same height.
lbs_OwnerDrawVariable	This style of list box must be drawn by its owner. Each entry may be a different height.
lbs_Sort	This style of list box sorts its entries alphabetically. The sort order may be different for list boxes with the *lbs_OwnerDrawFixed* or *lbs_OwnerDrawVariable* style.
lbs_Standard	This style is the same as the *lbs_Notify* and *lbs_Sort* styles together. The list box contains borders on all sides.
lbs_UseTabStops	This style of list box allows use of expanded *Tab* characters in its entries. By default there are tab stops each 32 dialog units from the left edge of the entry. A dialog unit is one-fourth of the dialog base width unit, which can be obtained by using the *GetDialogBaseUnits* function.
lbs_WantKeyboardInput	This style of list box has *wm_VKeyToItem* and *wm_CharToItem* messages sent to its owner when the list box has the input focus and a key is pressed.

lc_ Line capabilities

These constants represent the line rendering capabilities of a device.

Table 1.61
Line capability constants

Constant	Meaning
lc_None	Lines not supported.
lc_Interiors	Can do interiors.
lc_Marker	Can do markers.
lc_PolyLine	Can do polylines.
lc_PolyMarker	Can do polymarkers.
lc_Styled	Can do styled lines.
lc_Wide	Can do wide lines.
lc_WideStyled	Can do wide styled lines.

lf_FaceSize Logical font size constant

The *lf_FaceSize* constant indicates the number of bytes used to store the font face name in the *lfFaceName* field of *TLogFont* records. It is currently set to 32.

lmem_ Local memory flags

These flags specify characteristics of the local memory block created with the functions *LocalAlloc* and *LocalReAlloc*. *lmem_Discardable, lmem_Discarded, lmem_LockCount* are used in the *LocalFlags* function.

Table 1.62
Local memory flags

Constant	Meaning
lmem_Discardable	The memory block can be discarded. Must be used with *lmem_Moveable*.
lmem_Discarded	The memory block has been discarded. (*LocalFlags* only.)
lmem_LockCount	When combined with the low-order byte of the return value from *LocalFlags*, returns the memory block's reference count. (*LocalFlags* only.)
lmem_Fixed	The memory block is fixed in one memory location.
lmem_Modify	Modifies the *lmem_Discardable* flag.
lmem_Moveable	The memory block can move locations in memory.
lmem_NoCompact	When allocating the memory block, no other memory blocks will be compacted or discarded.
lmem_NoDiscard	When allocating the memory block, no other memory blocks will be discarded.
lmem_ZeroInit	Initializes the contents of the memory block to zero.

LPTx constant

LPTx is a bit mask for the *TDCB ID* field. If the bit mask is set, the device is an LPT (parallel port).

ma_ Mouse activation codes

These codes, when returned from a *wm_MouseActivate* message, indicates whether the window should be activated and whether the mouse event should be discarded.

Table 1.63
Mouse activation codes

Constant	Meaning
ma_Activate	Activate the window.
ma_ActivateAndEat	Activate the window; discard the mouse event.
ma_NoActivate	Do not activate the window.

mb_ Message box flags

These flags specify characteristics of the message box created with the *MessageBox* function. They are combined to create the desired style.

Table 1.64
Message box flags

Constant	Meaning
mb_AbortRetryIgnore	Include only Abort, Retry and Ignore buttons.
mb_ApplModal	Create a modal message box (the default).
mb_DefButton1	The default button is the first button (the default).
mb_DefButton2	The default button is the second button.
mb_DefButton3	The default button is the third button.
mb_IconAsterisk	Same as *mb_IconInformation*.
mb_IconExclamation	Include the '!' icon.
mb_IconHand	Same as *mb_IconStop*.
mb_IconInformation	Include the 'i' icon.
mb_IconQuestion	Include the '?' icon.
mb_IconStop	Include the stop sign icon.
mb_Ok	Include only an OK button.
mb_OkCancel	Include only OK and Cancel buttons.
mb_RetryCancel	Include only Retry and Cancel buttons.
mb_SystemModal	Create a modal message box that suspends Windows. Use this for potentially damaging situations.
mb_TaskModal	Use this flag if no parent window is available. Supply 0 for the parent parameter, and all top-level windows in the application will be suspended.
mb_YesNo	Include only Yes and No buttons.
mb_YesNoCancel	Include only Yes, No, and Cancel buttons.

Several bit masks are defined for groups of *mb_* constants:

Table 1.65
Message box flag masks

Mask	Constants masked
mb_DefMask	*mb_DefButton1, mb_DefButton2, mb_DefButton3*
mb_IconMask	*mb_IconAsterisk, mb_IconExclamation, mb_IconHand, mb_IconInformation, mb_IconQuestion, mb_IconStop*
mb_ModeMask	*mb_ApplModal, mb_SystemModal, mb_TaskModal*
mb_TypeMask	*mb_AbortRetryIgnore, mb_Ok, mb_OkCancel, mb_RetryCancel, mb_YesNo, mb_YesNoCancel*

meta_ Metafile codes

These constants correspond to particular GDI functions. For example, the value of *meta_Arc* is the numeric index of the *Arc* GDI function. The index values of these functions (and corresponding constants) can be stored in a Windows metafile, which is a list of GDI commands that can be played by a program to produce graphic output.

meta_AnimatePalette
meta_Arc
meta_BitBlt
meta_Chord
meta_CreateBitmap
meta_CreateBitmapIndirect
meta_CreateBrush
meta_CreateBrushIndirect
meta_CreateFontIndirect
meta_CreatePalette
meta_CreatePatternBrush
meta_CreatePenIndirect
meta_CreateRegion
meta_DeleteObject
meta_DIBBitBlt
meta_DIBCreatePatternBrush
meta_DIBStretchBlt
meta_DrawText
meta_Ellipse
meta_Escape
meta_ExcludeClipRect
meta_ExtTextOut
meta_FillRegion
meta_FloodFill
meta_FrameRegion
meta_IntersectClipRect
meta_InvertRegion
meta_LineTo
meta_MoveTo
meta_OffsetClipRgn
meta_OffsetViewportOrg
meta_OffsetWindowOrg
meta_PaintRegion
meta_PatBlt
meta_Pie
meta_Polygon
meta_PolyLine
meta_PolyPolygon
meta_RealizePalette
meta_Rectangle
meta_ResizePalette
meta_RestoreDC
meta_RoundRect
meta_SaveDC
meta_ScaleViewportExt
meta_ScaleWindowExt
meta_SelectClipRegion
meta_SelectObject
meta_SelectPalette
meta_SetBKColor

meta_SetBKMode
meta_SetDIBToDev
meta_SetMapMode
meta_SetMapperFlags
meta_SetPalEntries
meta_SetPixel
meta_SetPolyFillMode
meta_SetRelAbs
meta_SetROP2
meta_SetStretchBltMode
meta_SetTextAlign
meta_SetTextCharExtra
meta_SetTextColor
meta_SetTextJustification
meta_SetViewportExt
meta_SetViewportOrg
meta_SetWindowExt
meta_SetWindowOrg
meta_StretchBlt
meta_TextOut

mf_ Menu flags

The following constants are used as flags in many menu functions and in the *wm_MenuSelect* message.

Table 1.66
Menu flags

Constant	Meaning
mf_Bitmap	The menu item is a bitmap rather than a string.
mf_ByCommand	Indicates that the menu item will be specified with the menu item ID.
mf_ByPosition	Indicates that the menu item will be specified by position, where the first item is at position zero.
mf_Checked	Display a checkmark next to the menu item.
mf_Disabled	Disables the menu item.
mf_Enabled	Enables the menu item.
mf_Grayed	Disables and grays the menu item.
mf_Help	Indicates that the item is the help menu item.
mf_Hilite	Highlight the menu item.
mf_MenuBarBreak	Puts the popup menu item in a new column, separated by a bar.
mf_MenuBreak	Puts the popup menu item in a new column or the menu bar item in a new line.
mf_MouseSelect	Indicates that the item was selected with the mouse.
mf_OwnerDraw	Indicates that the menu item is an owner draw-item.
mf_Popup	Indicates that the new menu item also has sub-items.
mf_Separator	Inserts a horizontal separator bar in the menu.
mf_String	Indicates that the new menu item is a string.
mf_SysMenu	Indicates that the item is in the Control menu.
mf_Unchecked	Removes a checkmark, if any.
mf_Unhilite	Removes highlighting from the menu item.

mk_ Key state masks

These masks are combined and passed in mouse click messages to find out about the state of particular keys and mouse buttons.

Table 1.67
Key state masks

Constant	Meaning
mk_Control	The *Ctrl* key is down.
mk_LButton	The left mouse button is down.
mk_MButton	The middle mouse button is down.
mk_RButton	The right mouse button is down.
mk_Shift	The *Shift* key is down.

mm_ Mapping modes

These constants specify the device context's mapping mode, the method by which logical units are converted into device units as well as the orientation of the axes. They are used in the *GetMapMode* and *SetMapMode* functions.

Table 1.68
Mapping modes

Constant	Meaning
mm_Anisotropic	The units and axes are determined arbitrarily.
mm_HiEnglish	One logical unit equals 0.001 inches. The positive x-axis is to the right and the positive y-axis is up.
mm_HiMetric	One logical unit equals 0.01 millimeters. The positive x-axis is to the right and the positive y-axis is up.
mm_Isotropic	One logical x-axis unit equals one logical y-axis unit.
mm_LoEnglish	One logical unit equals 0.01 inches. The positive x-axis is to the right and the positive y-axis is up.
mm_LoMetric	One logical unit equals 0.1 millimeters. The positive x-axis is to the right and the positive y-axis is up.
mm_Text	One logical unit equals one pixel. The positive x-axis is to the right and the positive y-axis is down.
mm_TWips	One logical unit equals 1/1440 of an inch. The positive x-axis is to the right and the positive y-axis is up.

msgf_ Filter proc codes

These codes, passed in the *Code* parameter of message filter functions *wh_MsgFilter* and *wh_SysMsgFilter*, specify the types of messages processed.

Table 1.69 Messages processed with filter hooks

Constant	Messages processed	
	wh_MsgFilter	***wh_SysMsgFilter***
msgf_DialogBox	Message and dialog boxes	Dialog boxes only
msgf_Menu	Keyboard and mouse	Keyboard and mouse
msgf_MessageBox		Message boxes only

obj_ GDI object type constants

These two constants specify the type of GDI object, a pen or a brush, to be enumerated in a call to the function *EnumObjects*. They should be passed in the *ObjectType* parameter to *EnumObjects*.

Table 1.70 Object enumeration constants

Constant	Meaning
obj_Pen	Enumerate pens
obj_Brush	Enumerate brushes

obm_ Predefined bitmaps

These constants specify one of the many bitmaps Windows predefines for its own use. The programmer can use them with the *LoadBitmap* function. Those constants that begin with *obm_Old* refer to bitmaps used by Windows versions previous to 3.0.

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

oda_ Owner draw actions

These constants define an action to take place for an owner- draw control when used in the *itemAction* field of the record *TDrawItemStruct*. This field can be a combination of these constants.

Table 1.71 Owner-draw control action constants

Constant	Meaning
oda_DrawEntire	The entire control must be drawn.
oda_Focus	The focus was lost or gained.
oda_Select	The selection state has changed.

ods_ Owner draw states

These constants define the state of an owner-draw control after drawing when used in the *itemState* field of the record *TDrawItemStruct*. This field can be a combination of these constants.

Table 1.72 Owner-draw control state constants

Constant	Meaning
ods_Checked	Check the menu item.
ods_Disabled	Disable the item.
ods_Focus	Give the item the input focus.
ods_Grayed	Gray the menu item.
ods_Selected	Select the item.

odt_ Owner draw control types

These constants specify a particular type of owner-draw controls. They are used in the *CtlType* field of the records *TCompareItemStruct*, *TDrawItemStruct*, and *TMeasureItemStruct*.

Table 1.73 Owner-draw control type constants

Constant	Meaning
odt_Menu	Owner-draw menu
odt_ListBox	Owner-draw list box
odt_ComboBox	Owner-draw combo box
odt_Button	Owner-draw button

of_ Open file constants

These constants, when combined and passed in calls to the *OpenFile* or *_lopen* functions, specify a file opening action.

Table 1.74
Open file constants

Constant	Meaning
of_Cancel	Add a Cancel button to the dialog box produced with the *of_Prompt* constant. If Cancel is pressed, *OpenFile* returns a file-not-found error. (*OpenFile* only.)
of_Create	Open a new file or clear an existing one. (*OpenFile* only.)
of_Delete	Delete a file. (*OpenFile* only.)
of_Exist	Test a named file for existence. (*OpenFile* only.)
of_Parse	Only fill the *OFSTRUCT* record passed in the *ReOpenBuff* parameter. (*OpenFile* only.)
of_Prompt	If the file cannot be found, display a file-not-found dialog box. (*OpenFile* only.)
of_Read	Open a read-only file.
of_ReadWrite	Open a read-and-write file.
of_ReOpen	Open a file using the *OFSTRUCT* record passed in the *ReOpenBuff* parameter. (*OpenFile* only.)
of_Share_Compat	Open a file, but allow other processes to open the file many times.
of_Share_Deny_None	Open a file, but allow other processes to read from or write to the file.
of_Share_Deny_Read	Open a file, but prevent other processes from reading the file.
of_Share_Deny_Write	Open a file, but prevent other processes from writing to the file.
of_Share_Exclusive	Open a file, but prevent other processes from reading or writing to the file.
of_Verify	Verify that the time and date of a file is identical to when it was previously opened. (*OpenFile* only.)
of_Write	Open a write-only file.

out_ Font output precision flags

These constants specify the output precision of fonts produced with the *CreateFont* function.

Out_Default_Precis
Out_String_Precis
Out_Character_Precis
Out_Stroke_Precis

pc_ Palette entry flags

These flags specify information about a palette. They are used in the *peFlags* field of the *TPaletteEntry* record, and passed in calls to the *AnimatePalette*, *GetPaletteEntries*, *GetSystemPaletteEntries*, and *SetPaletteEntries* functions.

Table 1.75
Palette entry flags

Constant	Meaning
pc_Explicit	The low-order word of the logical palette entry specifies a hardware palette index.
pc_NoCollapse	The color will not be matched to existing palette color but will be made a new entry in the system palette.
pc_Reserved	The color will be used for palette animation and will change frequently.

pc_ Polygonal capabilities

These constants represent the polygon rendering capabilities of a device.

Table 1.76
Polygonal capability constants

Constant	Meaning
pc_Interiors	Can do interiors.
pc_None	Polygonals not supported.
pc_Polygon	Can do polygons.
pc_Rectangle	Can do rectangles.
pc_ScanLine	Can do scanlines.
pc_Styled	Can do styled borders.
pc_Trapezoid	Can do trapezoids.
pc_Wide	Can do wide borders.
pc_WideStyled	Can do wide styled borders.
pc_WindPolygon	Can do winding polygons.

pm_ Peek message options

These options specify the handling of messages processed with the *PeekMessage* function.

Table 1.77 *PeekMessage* options

Constant	Meaning
pm_NoRemove	After processing, messages are not removed from the queue.
pm_NoYield	Lets no other task interrupt the processing of the current task.
pm_Remove	After processing, messages are removed from the queue.

PolyFill modes

These constants specify the method in which complex polygons with more than one bounded regions are filled. Use them in calls to the *CreatePolygonRgn*, *CreatePolyPolygonRgn*, *GetPolyFillMode*, and *SetPolyFillMode* functions.

Table 1.78 PolyFill modes

Constant	Meaning
Alternate	Fills in alternating regions.
Winding	Fills in every region.

pr_ Spooler status code

pr_JobStatus is the value passed in *wParam* to the *wm_SpoolerStatus* message.

Printer escape codes

These codes are passed in the *Escape* function to allow applications to directly access device-specific facilities not supported by GDI.

Table 1.79 Printer escape codes

Constant	Action
AbortDoc	Terminates current printing job.
BandInfo	Copies information about the device's banding capabilities into a record passed in the call to *Escape*.

Table 1.79: Printer escape codes (continued)

Begin_Path	Opens a path, which is a polygon or polyline made of a connected sequence of primitives. Paths are used to simplify the instructions sent to PostScript printers.
Clip_To_Path	Defines a clipping region based on the currently open path.
DeviceData	Same as *PassThrough*.
DraftMode	Turns draft mode on or off.
DrawPatternRect	Draws a patterned, gray-scale or black rectangle on a PCL-compatible Hewlett-Packard printer.
EnableDuplex	Enables the ability of the printer to print on both sides of the paper.
EnablePairKerning	Enables or disables the printer's ability to kern character pairs.
EnableRelativeWidths	Enables or disables the relative character widths which, when disabled, guarantees that the extent of the string will equal the sum of the character extents. When enabled, you must access the font's extent table and compute string widths.
End_Path	Ends a path. See *Begin_Path*.
EndDoc	Ends a print job started by *StartDoc*. See *StartDoc*.
EnumPaperBins	Retrieves information about the printer's paper bins. Use *GetSetPaperBins* instead. This escape is provided only for backward compatiblity.
EnumPaperMetrics	Retrieves the number of supported paper types or size of the printable rectanges. Use the *ExtDeviceMode* function instead. This escape is provided only for backward compatiblity.
EPSPrinting	Suppresses the PostScript header control section, disallowing any GDI calls.
Ext_Device_Caps	Retrieves information about device-specific printer capabilities beyond what can be learned from calling the function *GetDeviceCaps*.
FlushOutput	Clears the device's buffer.
GDIExtTextOut	Performs the same task as the function *ExtTextOut*.
GDIStretchBlt	Performs the same task as the function *StretchBlt*.
GetColorTable	Retrieves an RGB color table entry.
GetExtendedTextMetrics	Retrieves the extended text metrics for the selected font.
GetExtentTable	Retrieves the extent (width) of a range of characters in the selected font's character set.
GetPairKernTable	Retrieves the character-pair kerning table.
GetPhysPageSize	Retrieves the physical page size.
GetPrintingOffset	Retrieves the offset from the upper-left corner of the page where the printing or drawing begins.
GetScalingFactor	Retrieves the x- and y-axis scaling factor of the printer.
GetSetPaperBins	Retrieves the number of printer paper bins and sets the current bin.
GetSetPaperMetrics	Retrieves the printer's paper metrics information and sets it with new information. Use the *ExtDeviceMode*

Table 1.79: Printer escape codes (continued)

	function instead. This escape is provided only for backward compatiblity.
GetSetPrintOrient	Retrieves or sets the current paper orientation. Use the *ExtDeviceMode* function instead. This escape is provided only for backward compatiblity.
GetTechnolgy	Retrieves a string identifying the printer's general technology, such as PostScript.
GetTrackKernTable	Retrieves the track-kerning table for the currently selected font.
GetVectorBrushSize	Retrieves the width in device units of the plotter's pen used to fill closed shapes.
GetVectorPenSize	Retrieves the width in device units of the plotter's pen used in hatched brush patterns.
MFComment	Adds a comment to a metafile.
NewFrame	Informs the printer to advance to a new page.
NextBand	Informs the device driver that output to a band is complete.
PassThrough	Allows the application to send data directly to the printer.
QueryEscSupport	Determines whether the device driver supports a particular escape.
Restore_CTM	Restores the previously saved current transformation matrix.
Save_Ctm	Saves the current transformation matrix.
SelectPaperSource	Superseded by *GetSetPaperBins*. Provided only for backward compatibility.
Set_Arc_Direction	Specifies the direction in which arcs are drawn using the *Arc* function.
Set_Background_Color	Sets and retrieves the device's background color.
Set_Bounds	Sets the bounding rectangle for the picture being produced.
Set_Clip_Box	Sets or restores the previous clipping rectangle.
Set_Poly_Mode	Sets the poly mode, which determines how to interpret calls to the *Polygon* and *PolyLine* functions.
Set_Screen_Angle	Sets the current screen angle to simulate the tilting of a phographic mask in producing a color separation.
Set_Spread	Sets the amount that nonwhite primitives are expanded to make up for imperfections in the reproduction process.
SetAbortProc	Sets the abort function for the currrent print job.
SetAllJustValues	Sets the justification values for text output.
SetColorTable	Sets an RGB color-table entry.
SetCopyCount	Specifies the number of copies of each page to print.
SetKernTrack	Specifies which kerning track to use.
SetLineJoin	Specifies how to join to intersecting lines: with a rounded, square or mitered corner.
SetMiterLimit	Sets the miter limit, which is the angle at which a miter join is replaced with a bevel join.
StartDoc	Informs the device driver that a new job is starting.
Transform_CTM	Modifies the current transformation matrix.

proc_ constants

These constants are passed as the *ProcName* parameter of the *GetProcAddress* function when the module handle passed in the *Module* parameter is a device driver.

proc_ExtDeviceMode
proc_DeviceCapabilities
proc_OldDeviceMode

ps_ Pen styles

The following constants specify styles for pen tools. They are used in the *CreatePen* function.

Table 1.80
Pen styles

Constant	Meaning
ps_Solid	———
ps_Dash	- - - -
ps_Dot	. . . .
ps_DashDot	_·_·_·_
ps_DashDotDot	_··_··_
ps_Null	Invisible.
ps_InsideFrame	Used to draw the inside frame of polygons and polylines.

r2_ Binary raster operations

These constants specify the drawing mode for combining the colors of the current pen with the existing colors. They are used as parameters in the *GetROP2* and *SetROP2* functions.

Table 1.81
Binary raster operation constants

Constant	Meaning
r2_Black	The resulting color is always black.
r2_CopyPen	The resulting color is always the pen color.
r2_MaskNotPen	The resulting color is a combination of the colors common to the existing display and the inverse of the current pen.
r2_MaskPen	The resulting color is a combination of the colors common to both the existing display and the current pen.

Table 1.81: Binary raster operation constants (continued)

r2_MaskPenNot	The resulting color is the inverse of the color resulting from *r2_MaskPen*.
r2_MergeNotPen	The resulting color is a combination of the inverse of the current pen color and the existing display color.
r2_MergePen	The resulting color is a combination of the current pen color and the existing display color.
r2_MergePenNot	The resulting color is a combination of the current pen color and the inverse of the existing display color.
r2_Nop	The existing display does not change.
r2_Not	The resulting color is the inverse of the existing display color.
r2_NotCopyPen	The resulting color is the inverse of the current pen color.
r2_NotMaskPen	The resulting color is the inverse of the color resulting from the *r2_MaskPen* mode.
r2_NotMergePen	The resulting color is the inverse of the color resulting from the *r2_MergePen* mode.
r2_NotXorPen	The resulting color is the inverse of the color resulting from the *r2_XorPen* mode.
r2_White	The resulting color is white.
r2_XorPen	The resulting color is the combination of the colors in the existing display and the current pen, but not in both.

rc_ Raster capabilities

These constants represent the raster capabilities of a device.

Table 1.82
Raster capability constants

Constant	Meaning
rc_Banding	Device requires banding support.
rc_BigFont	Supports larger than 64K fonts.
rc_BitBlt	Can transfer bitmaps.
rc_Bitmap64	Device can support larger than 64K bitmaps.
rc_DI_Bitmap	Supports DIB to memory.
rc_DIBToDev	Supports *DIBitsToDevice* function.
rc_FloodFill	Supports the *FloodFill* function.
rc_GDI20_Output	Device supports Windows 2.0 capabilities.
rc_Palette	Supports a palette.
rc_Scaling	Device requires scaling support.
rc_StretchBlt	Supports the *StretchBlt* function.
rc_StretchDIB	Supports the *StretchDIBits* function.

Region flags

These flags specify a type of graphic region. They are used as return values from *CombineRgn*, *ExcludeClipRect*, *ExcludeUpdateRgn*, *GetClipBox*, *GetRgnBox*, *GetUpdateRgn*, *OffsetClipRgn*, *OffsetRgn*, and *SelectClipRgn* functions.

Table 1.83 Graphic region flags

Constant	Meaning
ComplexRegion	The region has overlapping borders.
Error	No new region was created.
NullRegion	The region is empty.
SimpleRegion	The region has no overlapping borders.

Resource type constant

This value is the number of defined resource types minus one. If you add a new type, you must increment *Difference*, initially defined as 11.

rgn_ Combine region flags

These flags specify the combination method used by the *CombineRgn* function to combine two regions.

Table 1.84 Combine region flags

Constant	Meaning
rgn_And	The resulting region is the intersection of the two provided regions.
rgn_Copy	The resulting region is a copy of the first provided region.
rgn_Diff	The resulting region includes the parts of the first region that are not also in the second region.
rgn_Or	The resulting region is the union of the two provided regions.
rgn_Xor	The resulting region includes the parts of the each region that are not in both regions.

rt_ Resource types

These constants represent the available types of Windows resources. They are used as parameters in the *FindResource* function.

Table 1.85
Resource type constants

Constant	Meaning
rt_Accelerator	Accelerator table.
rt_Bitmap	Bitmap resource.
rt_Cursor	Cursor resource.
rt_Dialog	Dialog box template.
rt_Font	Font resource.
rt_FontDir	Font directory resource.
rt_Icon	Icon resource.
rt_Menu	Menu resource.
rt_RcData	User-defined resource (raw data).
rt_String	String resource.

s_ Sound constants

These constants are used in a variety of sound functions, and are categorized by the function in which they are used.

Error values

These negative error values are returned from the sound functions.

Table 1.86
Sound constants

Constant	Meaning
s_SerBDNT	Invalid note.
s_SerDCC	Invalid note count.
s_SerDDR	Invalid duration.
s_SerDFQ	Invalid frequency.
s_SerDLN	Invalid note length.
s_SerDMD	Invalid mode.
s_SerDPT	Invalid pitch.
s_SerDSH	Invalid shape.
s_SerDSR	Invalid source.
s_SerDST	Invalid state.
s_SerDTP	Invalid tempo.
s_SerDVL	Invalid volume
s_SerDVNA	Device not available.
s_SerMACT	Music active.
s_SerOFM	Out of memory.
s_SerQFUL	Queue full.

SetSoundNoise

These constants specify the noise source, where *N* is a value used to derive a target frequency.

Table 1.87
Noise source constants

Constant	Meaning
s_Period512	High pitch, N/512. Hiss is less coarse.
s_Period1024	Medium pitch, N/1024.
s_Period2048	Low pitch, N/2048. Hiss is more coarse.
s_PeriodVoice	From voice channel 3.
s_White512	High pitch, N/512. Hiss is less coarse.
s_White1024	Medium pitch, N/1024.
s_White2048	Low pitch, N/2048. Hiss is more coarse.
s_WhiteVoice	From voice channel 3.

SetVoiceAccent

These constants specify the accent of the notes to be played.

Table 1.88
Note accent options

Constant	Meaning
s_Legato	Hold the note for the full duration, blending it with the start of the next note.
s_Normal	Hold the note for the full duration, coming to a stop before the start of the next note.
s_Staccato	Hold the note for shorter than the full duration, with a noticeable pause before the next note.

WaitSoundState

These constants specify the state of the voice queues.

Table 1.89
Voice queue options

Constant	Meaning
s_AllThreshold	All voice queues have reached threshold.
s_QueueEmpty	All voice queues are empty and sound drivers are off.
s_Threshold	One voice queue has reached threshold and returns voice.

sb_ Scroll bar commands

These constants specify scroll bar events and are passed as parameters in *wm_HScroll*, *wm_HScrollClipboard*, *wm_VScroll*, and *wm_VScrollClipboard* messages.

Table 1.90
Scroll bar event constants

Constant	Meaning
sb_Bottom	A scroll bar has been scrolled down or to the right.
sb_EndScroll	A scroll bar has been scrolled to the end.
sb_LineDown	A scroll bar has been scrolled one line down.
sb_LineUp	A scroll bar has been scrolled one line up.
sb_PageDown	A scroll bar has been scrolled one page down.

Table 1.90: Scroll bar event constants (continued)

sb_PageUp	A scroll bar has been scrolled one page up.
sb_ThumbPosition	The thumb has been dragged to an absolute position.
sb_ThumbTrack	The thumb has been dragged to a specified position.
sb_Top	A scroll bar has been scrolled up or to the left.

sb_ Scroll bar constants

These constants identify the particular scroll bar specified in the *GetScrollPos*, *GetScrollRange*, *SetScrollPos*, *SetScrollRange*, and *ShowScrollBar* functions.

Table 1.91
Scroll bar constants

Constant	Meaning
sb_Both	Specifies the window's horizontal and vertical scroll bars. *sb_Both* is used only in the *ShowScrollBar* function.
sb_Ctl	Specifies a standalone scroll bar control.
sb_Horz	Specifies the window's horizontal scroll bar.
sb_Vert	Specifies the window's vertical scroll bar.

sbs_ Scroll bar styles

These constants are used to specify scroll bar styles when creating scroll bars with the *CreateWindow* and *CreateWindowEx* functions.

Table 1.92
Scroll bar styles

Constant	Meaning
sbs_BottomAlign	This style of scroll bar is of default height and has its bottom edge aligned to the bottom edge of the rectangle used to create it. The *sbs_BottomAlign* style may only be used if the *sbs_Horz* style is used also.
sbs_Horz	This style of scroll bar is horizontal. If neither the *sbs_BottomAlign* nor the *sbs_TopAlign* style is used, the scroll bar will be the exact size requested when it is created.
sbs_LeftAlign	This style of scroll bar is of default width and has its left edge aligned to the left edge of the rectangle used to create it. The *sbs_LeftAlign* style may only be used if the *sbs_Vert* style is used also.
sbs_RightAlign	This style of scroll bar is of default width and has its right edge aligned to the right edge of the rectangle used to create it. The

Table 1.92: Scroll bar styles (continued)

	sbs_RightAlign style may only be used if the *sbs_Vert* style is used also.
sbs_SizeBox	This style of scroll bar is a size box. If neither the *sbs_SizeBoxBottomRightAlign* nor the *sbs_SizeBoxTopLeftAlign* style is used, the scroll bar will be the exact size requested when it is created.
sbs_SizeBoxBottomRightAlign	This style of scroll bar is of default size for system size boxes and has its lower-right corner aligned to the lower-right corner of the rectangle used to create it. The *sbs_SizeBoxBottomRightAlign* style may only be used if the *sbs_SizeBox* style is used also.
sbs_SizeBoxTopLeftAlign	This style of scroll bar is of default size for system size boxes and has its upper-left corner aligned to the upper-left corner of the rectangle used to create it. The *sbs_SizeBoxTopLeftAlign* style may only be used if the *sbs_SizeBox* style is used also.
sbs_TopAlign	This style of scroll bar is of default height and has its top edge aligned to the top edge of the rectangle used to create it. The *sbs_TopAlign* style may only be used if the *sbs_Horz* style is used also.
sbs_Vert	This style of scroll bar is vertical. If neither the *sbs_RightAlign* nor the *sbs_LeftAlign* style is used, the scroll bar will be the exact size requested when it is created.

sc_ System command values

The following commands are passed in the *wm_SysCommand* message in response to a selection in the Control menu or a click in the minimize or maximize box. They indicate the action requested.

Table 1.93 System command constants

Constant	Meaning
sc_Close	Close the window.
sc_HScroll	Horizontally scroll.
sc_KeyMenu	Get the menu through a key stroke.
sc_Maximize	Maximize the window.
sc_Minimize	Minimize the window.
sc_MouseMenu	Get the menu through a mouse click.
sc_Move	Move the window.
sc_NextWindow	Move to the next window.
sc_PrevWindow	Move to the previous window.

Table 1.93: System command constants (continued)

sc_Restore	Restore the window from a maximized or minimized state.
sc_Size	Size the window.
sc_TaskList	Bring up a task list.
sc_VScroll	Vertically scroll.

show_ Old ShowWindow commands

These are old constants, included only for backwards compatibility with previous versions of Windows. The *sw_* Show window constants should be used instead.

Table 1.94 Old ShowWindow Commands

Constant	Value	Meaning
hide_Window	0	Hide window
show_OpenWindow	1	Restore window from icon
show_IconWindow	2	Shrink window to icon
show_FullScreen	3	Zoom window to fill screen
show_OpenNoActivate	4	Restore window, but don't make it active

size Size constants

These constants indicate the type of window sizing occurred. They are passed in *wm_Size* messages.

Table 1.95 Window size constants

Constant	Meaning
SizeFullScreen	The window was maximized.
SizeIconic	The window was minimized.
SizeNormal	The window was resized, but not maximized or minimized.
SizeZoomHide	Some other window was maximized.
SizeZoomShow	Some other window was restored from a maximized state.

sm_ System metrics codes

These codes specify aspects of the Windows user interface about which the programmer can obtain dimensions with the *GetSystemMetrics* function.

Table 1.96
System metrics codes

Constant	Meaning
sm_CXBorder	Non-sizable window frame width.
sm_CXCursor	Cursor width.
sm_CXDlgFrame	*ws_DlgFrame* style window frame width.
sm_CXFrame	Sizable window frame width.
sm_CXFullScreen	Maximized window's client area width.
sm_CXHScroll	Horizontal scroll bar arrow width.
sm_CXHThumb	Horizontal scroll bar thumb width.
sm_CXIcon	Icon width.
sm_CXMin	Window's minimum width.
sm_CXMinTrack	Window's minimum tracking width.
sm_CXScreen	Screen width.
sm_CXSize	Title bar bitmap width.
sm_CXVScroll	Vertical scroll bar arrow width.
sm_CYBorder	Non-sizable window frame height.
sm_CYCaption	Caption height.
sm_CYCursor	Cursor height.
sm_CYDlgFrame	*ws_DlgFrame* style window frame height.
sm_CYFrame	Sizable window frame height.
sm_CYFullScreen	Maximized window's client area height.
sm_CYHScroll	Horizontal scroll bar arrow height.
sm_CYIcon	Icon height.
sm_CYKanjiWindow	Kanji window height.
sm_CYMenu	Single-line menu height.
sm_CYMin	Window's minimum height.
sm_CYMinTrack	Window's minimum tracking height.
sm_CYScreen	Screen height.
sm_CYSize	Title bar bitmap height.
sm_CYVScroll	Vertical scroll bar arrow height.
sm_CYVTHUMB	Vertical scroll bar thumb width.
sm_Debug	Return zero if the version of Windows running is not the debugging version.
sm_MousePresent	Return zero if mouse is absent.
sm_SwapButton	Return zero if mouse buttons are not swapped.

S

sp_ Spooler error codes

These constants are returned as error codes from the *Escape* function.

Table 1.97
Spooler error codes

Constant	Meaning
sp_AppAbort	The application terminated the job.
sp_Error	General error.
sp_OutOfDisk	Not enough disk space for spooling.
sp_OutOfMemory	Not enough memory for spooling.
sp_UserAbort	User terminated the job from the spooler.

ss_ Static control styles

These constants are used to specify static control styles when creating static controls with the *CreateWindow* and *CreateWindowEx* functions.

Table 1.98
Static control styles

Constant	Meaning
ss_BlackFrame	This style of static control has a frame with the same color as window frames.
ss_BlackRect	This style of static control is filled with the same color used to draw window frames.
ss_Center	This style of static control displays its text centered in its rectangle. If the text is longer than will fit in the width of the control, the line is wrapped to the next line. Lines are broken at word boundaries and each line is centered.
ss_GrayFrame	This style of static control has a frame with the same color as the screen background.
ss_GrayRect	This style of static control is filled with the color used to fill the screen background.
ss_Icon	This style of static control is an icon. The text of the control is the name of the icon as found in the resource file. Icons automatically size themselves.
ss_Left	This style of static control displays its text flush-left in its rectangle. If the text is longer than will fit in the width of the control, the line is wrapped to the next line. Lines are broken at word boundaries and each line is left aligned.
ss_LeftNoWordWrap	This style of static control displays its text flush-left in its rectangle. If the text is longer than will fit in the width of the control, the extra text is clipped.
ss_NoPrefix	This style of static control ignores '&' characters in its text. Normally the '&' character is used as an accelerator prefix character which is removed and the next character in the string is underlined.

Table 1.98: Static control styles (continued)

ss_Right	This style of static control displays its text flush-right in its rectangle. If the text is longer than will fit in the width of the control, the line is wrapped to the next line. Lines are broken at word boundaries and each line is right aligned.
ss_Simple	This style of static control displays one line of text flush-left. The text cannot be altered. The control's parent must not process the *wm_CtlColor* message.
ss_UserItem	This style of static control is a user defined static control.
ss_WhiteFrame	This style of static control has a frame with the same color as window backgrounds.
ss_WhiteRect	This style of static control is filled with the color used to fill window backgrounds.

Stock logical objects

These constants represent predefined (stock) drawing tools. They are used in the *GetStockObject* function.

Table 1.99
Stock logical object constants

Constant	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-pitch system font
ANSI_Var_Font	ANSI variable-pitch system font
Device_Default_Font	Device-dependent font
OEM_Fixed_Font	OEM-dependent fixed font
System_Fixed_Font	Fixed-pitch font from previous versions of Windows
System_Font	The Windows system font (variable-pitch)
Default_Palette	The default color palette

StretchBlt modes

These constants are bitmap stretching modes used in *GetStretchBltMode* and *SetStretchBltMode* functions.

Table 1.100
Bitmap stretching modes

Constant	Meaning
BlackOnWhite	Use the **AND** operator to eliminate the bitmap's lines, preserving the bitmap's black sections.
ColorOnColor	Eliminates lines regardless of their content, losing information in the process.
WhiteOnBlack	Use the **OR** operator to eliminate the bitmap's lines, preserving the bitmap's white sections.

sw_ Show window constants

These constants indicate the state in which the *ShowWindow* function displays a window.

Table 1.101
ShowWindow constants

Constant	Meaning
sw_Hide	Hidden.
sw_Maximize	Same as *sw_ShowMaximized*.
sw_Minimize	Minimized and inactive.
sw_Normal	Same as *sw_ShowNormal*.
sw_OtherZoom	Another window is being maximized. (Included for Windows 2.0 compatibility.)
sw_OtherUnzoom	Another window is being minimized. (Included for Windows 2.0 compatibility.)
sw_Restore	Same as *sw_ShowNormal*.
sw_Show	In the window's current size and position.
sw_ShowMaximized	Maximized and active.
sw_ShowMinimized	Minimized and active.
sw_ShowMinNoActive	Minimized. Does not affect window activation.
sw_ShowNA	In the window's current state. Does not affect window activation.
sw_ShowNoActivate	In the window's current size and position. Does not affect window activation.
sw_ShowNormal	Restored and active.

sw_ Show window message constants

These identifiers indicate the status of the window being shown and are passed in the *wm_ShowWindow* message.

Table 1.102
wm_ShowWindow constants

Constant	Meaning
sw_ParentClosing	Either the window's parent is being minimized or a popup window is being hidden.
sw_ParentOpening	Either the window's parent window is being displayed or a popup window is being shown.

swp_ Set window position flags

These flags are passed in the *SetWindowPos* and *DeferWindowPos* functions to signify one or many actions relating to the specified window.

Table 1.103
Set window position flags

Constant	Meaning
swp_DrawFrame	Draws a window frame defined by the window class.
swp_HideWindow	Hides the window.
swp_NoActivate	Does not affect the active window.
swp_NoMove	Does not reposition the window according to the supplied coordinates.
swp_NoRedraw	Does not redisplay the window to reflect the requested changes.
swp_NoSize	Does not resize the window according to the supplied width and height.
swp_NoZOrder	Does not affect the ordering of the windows.
swp_ShowWindow	Displays the window.

syspal_ System palette flags

These constants specify the new use of the system palette. They are used in the *GetSystemPaletteUse* and *SetSystemPaletteUse* functions.

Table 1.104
System palette flags

Constant	Meaning
syspal_NoStatic	The system palette contains no static colors besides black and white.
syspal_Static	The system palette contains static colors that will not change when an application realizes its logical palette.

ta_ Text alignment options

These alignment options govern the alignment of text drawn with the *TextOut* and *ExtTextOut* functions. They are specified or returned in the *GetTextAlign* and *SetTextAlign* functions.

Table 1.105
Text alignment options

Constant	Meaning
ta_BaseLine	By the baseline of the current font.
ta_Bottom	By the bottom of the bounding rectangle.
ta_Center	By the horizontal center of the bounding rectangle.
ta_Left	By the left side of the bounding rectangle. This is a default.
ta_NoUpdateCP	The current position is not updated after each call to *TextOut* or *ExtTextOut*. This is a default.
ta_Right	By the right side of the bounding rectangle.
ta_Top	By the top of the bounding rectangle. This is a default.
ta_UpdateCP	The current position is updated after each call to *TextOut* or *ExtTextOut*.

tc_ Text capabilities

These constants represent the text rendering capabilities of a device.

Table 1.106
Text capability constants

Constant	Meaning
tc_cp_Stroke	Can do stroke clip precision.
tc_cr_90	Can do 90-degree character rotation.
tc_cr_Any	Can do any character rotation.
tc_ea_Double	Can do double-weight characters.
tc_ia_Able	Can do italic characters.
tc_op_Character	Can do character output precision.
tc_op_Stroke	Can do stroke output precision.
tc_ra_Able	Can do raster fonts.
tc_sa_Contin	Can do any multiples for scaling.
tc_sa_Double	Can do doubled character for scaling.
tc_sa_Integer	Can do integer multiples for scaling.
tc_sf_X_YIndep	Can do scaling independent of X and Y.
tc_so_Able	Can do striked-out characters.
tc_ua_Able	Can do underlined characters.
tc_va_Able	Can do vector fonts.

tf_ForceDrive GetTempFileName flag

When combined with the *DriveLetter* argument in a call to the *GetTempFileName* function, *tf_ForceDrive* ensures that the temporary file will be created on the specified drive. Otherwise, it will be created on the first hard disk or in the directory specified in the TEMP environment variable.

Ternary raster operations

The following constants are used as raster operation codes in the *BitBlt*, *PatBlt*, *StretchBlt*, and *StretchDIBits* functions.

Table 1.107
Ternary raster operation constants

Constant	Meaning
Blackness	Produces completely black output.
DSTInvert	Produces an inverted bitmap.
MergeCopy	Combines the pattern and the source bitmaps using the logical **AND** operation.
MergePaint	Combines the destination and the inverted source bitmaps using the logical **OR** operation.
NotSrcCopy	Inverts the source bitmap and copies it to the destination bitmap.
NotSrcErase	Inverts the result of *MergePaint*.
PatCopy	Copies the specified pattern to the bitmap.
PatInvert	Combines the destination bitmap and pattern using the logical **XOR** operation.
PatPaint	Combines the source bitmap and pattern using **OR**, and then with the destination bitmap using **OR**.
SrcAnd	Combines the source and destination bitmaps using the **AND** operation.
SrcCopy	Copies the source bitmap to the destination.
SrcErase	Combines the source and the inverted destination using **AND**.
SrcInvert	Combines the source and destination bitmaps using **XOR**.
SrcPaint	Combines the source and destination bitmaps using **OR**.
Whiteness	Produces completely white output.

vk_ Virtual key codes

Virtual key codes are constants used to represent a computer's standard keys, such as the letter 'A' or the *F1* key. Since different brands of computers might have a different set of keys, the virtual key codes are used when processing keyboard input enabling one application to run on many different computers. These codes are used when defining accelerators or when processing the keyboard messages: *wm_KeyDown*, *wm_KeyUp*, *wm_SysKeyDown*, and *wm_SysKeyUp*. The following table lists the virtual key codes and their corresponding keys.

Table 1.108 Standard virtual key set

Code	Key or button
vk_LButton	Left mouse button
vk_RButton	Right mouse button
vk_Cancel	Used for control-break processing
vk_MButton	Middle mouse button
vk_Back	Backspace
vk_Tab	Tab
vk_Clear	Clear
vk_Return	Return
vk_Shift	Shift
vk_Control	Ctrl
vk_Menu	Menu key
vk_Pause	Pause
vk_Capital	Caps Lock
vk_Escape	Esc
vk_Space	Spacebar
vk_Prior	Page Up
vk_Next	Page Down
vk_End	End
vk_Home	Home
vk_Left	Left arrow
vk_Up	Up arrow
vk_Right	Right arrow
vk_Down	Down arrow
vk_Select	Select
vk_Print	OEM specific
vk_Execute	Execute
vk_SnapShot	Printscreen
vk_Insert	Insert
vk_Delete	Delete
vk_Help	Help
vk_A to *vk_Z*	'A' to 'Z'
vk_0 to *vk_9*	'0' to '9'
vk_NumPad0 to *vk_NumPad0*	Numeric key pad '0' to '9'
vk_Multiply	Multiply (gray '*')
vk_Add	Add (gray '+')
vk_Separator	Separator

Table 1.108: Standard virtual key set (continued)

vk_Subtract	Subtract (gray '-')
vk_Decimal	Decimal (numeric key pad '.')
vk_Divide	Divide (gray '/')
vk_F1 to *vk_F16*	*F1* to *F16* function keys
vk_NumLock	*Num Lock*

wep_ DLL exit codes

These codes are used in the *ExitCode* variable accessible from a dynamic-link library.

Table 1.109
DLL exit codes

wep_Free_DLL	Indicates that only the DLL is terminating and will be removed from memory.
wep_System_Exit	Indicates that Windows itself is terminating.

wf_ Windows memory configuration flags

These constants specify the current Windows memory configuration. They are used as return values from the *GetWinFlags* function.

Table 1.110
Windows memory configuration flags

Constant	Meaning
wf_80x87	The computer has an Intel math co-processor.
wf_CPU086	The computer's CPU is an 8086.
wf_CPU186	The computer's CPU is an 80186.
wf_CPU286	The computer's CPU is an 80286.
wf_CPU386	The computer's CPU is an 80386.
wf_CPU486	The computer's CPU is an 80486.
wf_Enhanced	Windows is running in 386-Enhanced (protected) mode.
wf_LargeFrame	The system is configured as large frame.
wf_PMode	Windows is running under protected (386-Enhanced or Standard) mode.
wf_SmallFrame	The system is configured as small frame.
wf_Standard	Windows is running in Standard (protected) mode.
wf_Win286	Same as *wf_Standard*, or Standard mode.
wf_Win386	Same as *wf_Enhanced*, or 386 Enhanced mode.

wh_ Windows hook codes

These codes specify a particular type of filter function to be installed (*SetWindowsHook* function) or removed (*UnhookWindowsHook* function) from a chain of filter functions.

Table 1.111
Windows hook codes

Constant	Meaning
wh_CallWndProc	Window function filter.
wh_GetMessage	Message filter.
wh_JournalPlayback	Journaling playback filter.
wh_JournalRecord	Journaling record filter.
wh_Keyboard	Keyboard filter.
wh_MsgFilter	Message filter (*SetWindowsHook* only).
wh_SysMsgFilter	System-wide message filter (*SetWindowsHook* only).

ws_ Window styles

These constants are used in combination to specify window styles when creating windows, dialogs and controls with the *CreateWindow* and *CreateWindowEx* functions.

Table 1.112
Window styles

Constant	Meaning
ws_Border	This style of window has a border. The *ws_Border* style cannot be used with the *ws_DlgFrame* style.
ws_Caption	This style of window has a title bar and a border. The *ws_Caption* and *ws_DlgFrame* styles cannot be used together. The *ws_Border* style is implied when the *ws_Caption* style is used.
ws_Child	This style of window is a child window. The *ws_Child* and *ws_Popup* styles cannot be used together.
ws_ChildWindow	Same as *ws_Child*.
ws_ClipChildren	This style of window does not include the area covered by its child windows when drawing.
ws_ClipSiblings	This style of window clips all sibling windows when drawing. This means that drawable regions in each client region of child windows of the same parent will not overlap. If the *ws_ClipSiblings* style is used, the *ws_Child* style must be used also.
ws_Disabled	This style of window is initially disabled.
ws_DlgFrame	This style of window has a double border and no title. The *ws_DlgFrame* style cannot be used with the *ws_Border* style.
ws_Group	This style of window is a control which is the first of a group of controls which can be accessed using the arrow keys. Each control defined without the *ws_Group*

Table 1.112: Window styles (continued)

	style belongs to the group started by the last control with the *ws_Group* style.
ws_HScroll	This style of window has a horizontal scroll bar.
ws_Iconic	Same as *ws_Minimize*.
ws_Maximize	This style of window appears full screen (maximized).
ws_MaximizeBox	This style of window has a maximize box.
ws_Minimize	This style may only be used with the *ws_Overlapped* style. This style of window is initially minimized (iconic).
ws_MinimizeBox	This style of window has a minimize box.
ws_Overlapped	This style of window is an overlapped window. An overlapped window has a caption and a border.
ws_OverlappedWindow	This is the same as the *ws_Overlapped*, *ws_Caption*, *ws_SysMenu*, *ws_ThickFrame*, *ws_MinimizeBox*, and *ws_MaximizeBox* styles combined.
ws_Popup	This style of window is a pop-up window. The *ws_Child* and *ws_Popup* styles may not be used together.
ws_PopupWindow	This style is the same as the *ws_Border*, *ws_Popup*, and *ws_SysMenu* styles combined. The Control-menu box will be visible only if the *ws_Caption* style is used also.
ws_SizeBox	Same as *ws_ThickFrame*.
ws_SysMenu	This style of window has a Control-menu box in its title bar. It applies only to windows with title bars.
ws_TabStop	This style of window is a control which is in a list of controls which can be cycled through using the *Tab* key. The control must be the child of a dialog box.
ws_ThickFrame	This style of window has a thick frame that can be used to size the window.
ws_Tiled	Same as *ws_Overlapped*.
ws_TiledWindow	Same as *ws_OverlappedWindow*.
ws_Visible	This style of window is initially invisible.
ws_VScroll	This style of window has a vertical scroll bar.

ws_ex_ Extended window styles

These constants are used in combination with *ws_* constants to specify window extended window styles when creating windows with the *CreateWindowEx* function.

Table 1.113
Extended window styles

Constant	Meaning
ws_ex_DlgModalFrame	This style of window has a double border. The *ws_Caption* style may be used with the *ws_ex_DlgModalFrame* style.
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.

C H A P T E R

2

Windows function reference

This chapter provides a quick lookup guide for programmers who require a particular fact about a Windows API feature. It lists all Windows procedures and functions alphabetically.

For each procedure or function, you will find the following information. It provides the following information for each function: the header of the procedure or function, as declared in the unit *WinProcs*, a definition of each parameter, the return value, and a brief description of the operation the function performs.

Sample

Declaration `function Sample(AParam: PType; A2ndParam: P2ndType): Word;`

This is a brief description of the use of the *Sample* function.

Parameters *AParam*: This tells you what the first parameter is for.

A2ndParam: This explains the second parameter.

Returns The *Word* value returned by *Sample* means one thing if it has a certain value, and something else if it has another value.

See also *Unsample, Resample*

AccessResource

Declaration `function AccessResource(Instance, ResInfo: THandle): Integer;`

Opens and positions resource file to the beginning of a resource. The file should be closed after reading the resource.

Parameters *Instance*: Instance module whose executable file contains the resource.

ResInfo: Desired resource, created by calling the *FindResource* function.

Returns DOS file handle; –1 if resource not found.

See also *FindResource*.

AddAtom

Declaration `function AddAtom(Str: PChar): TAtom;`

Adds *Str* to the atom table. A reference count is maintained for each unique string instance.

Parameters *Str*: Null-terminated character string.

Returns Unique atom identifier if successful; –1 if not.

See also *GetAtomName*.

AddFontResource

Declaration `function AddFontResource(FileName: PChar): Integer;`

Adds font resource from *FileName* font-resource file to the system font table.

Parameters *FileName*: Handle to loaded module or null-terminated string.

Returns Number of fonts added; zero if no fonts added.

See also *wm_FontChange*.

AdjustWindowRect

Declaration `procedure AdjustWindowRect(var Rect: TRect; Style: Longint; Menu: Bool);`

Computes the required size of a window rectangle based on *Rect* size. Assumes a single row menu, if one.

Parameters *Rect*: *TRect* containing client rectangle coordinates to be converted.

Style: Window styles of the window whose client rectangle is to be converted.

Menu: Non-zero if window has a menu.

See also *CreateWindow*.

AdjustWindowRectEx

Declaration `procedure AdjustWindowRectEx(var Rect: TRect; Style: Longint; Menu: Bool; ExStyle: Longint);`

Computes the required size of a window rectangle with extended style based on *Rect* size. Assumes a single row menu, if one.

Parameters *Rect*: *TRect* structure containing client rectangle coordinates to be converted.

Style: Window styles of the window whose client rectangle is to be converted.

Menu: Non-zero if window has a menu.

ExStyle: Extended style of the window being created.

See also *CreateWindowEx*.

AllocDStoCSAlias

Declaration `function AllocDStoCSAlias(Selector: Word): Word;`

Maps *Selector* to a code-segment selector.

Parameters *Selector*: Data-segment selector.

Returns Corresponding code-segment selector if successful; zero if not.

AllocResource

Declaration `function AllocResource(Instance, ResInfo: THandle; Size: Longint): THandle;`

Allocates uninitialized memory for *ResInfo*.

Parameters *Instance*: Module instance of executable file containing the resource.

ResInfo: Desired resource.

Size: Size in bytes to allocate for the resource; ignored if zero.

Returns Allocated global memory block.

See also *FindResource, LoadResource.*

AllocSelector

Declaration `function AllocSelector(Selector: Word): Word;`

Allocates a new selector which is an exact copy of *Selector*. If *Selector* is **nil** allocates a new, uninitialized selector.

Parameters *Selector*: Selector to be copied.

Returns A selector if successful; zero if not.

AnimatePalette

Declaration `procedure AnimatePalette(Palette: HPalette; StartIndex: Word; NumEntries: Word; var PaletteColors);`

Replaces entries in *Palette* between *StartIndex* and *NumEntries* with *PaletteColors*.

Parameters *Palette*: The logical palette.

StartIndex: The first entry in palette to be animated.

NumEntries: The number of entries in palette to be animated.

PaletteColors: An array of *TPaletteEntry* structures.

See also *CreatePalette*.

AnsiLower

Declaration `function AnsiLower(Str: PChar): PChar;`

Uses language driver to convert *Str* to lowercase.

Parameters *Str*: Null-terminated string or a single character (in low-order byte).

Returns Converted string or character.

AnsiLowerBuff

Declaration `function AnsiLowerBuff(Str: PChar; Length: Word): Word;`

Uses language driver to convert *Str* to lowercase.

Parameters *Str*: Buffer of characters.

Length: Number of characters in buffer; if zero length is 64K (65,536).

Returns Length of the converted string.

AnsiNext

Declaration `function AnsiNext(CurrentChar: PChar): PChar;`

Used to iterate through strings whose characters are two or more bytes long.

Parameters *CurrentChar*: Null-terminated string.

Returns Pointer to next character in string.

AnsiPrev

Declaration `function AnsiPrev(Start, CurrentChar: PChar): PChar;`

Used to iterate backwards through strings whose characters are two or more bytes long.

Parameters *Start*: Beginning of string (null-terminated).

CurrentChar: Points to a character in the string.

Returns Pointer to previous character in the string.

AnsiToOem

Declaration `function AnsiToOem(AnsiStr, OemStr: PChar): Integer;`

Translates *AnsiStr* to OEM-defined character set. Length can be greater than 64K.

Parameters *AnsiStr*: String (null-terminated) of ANSI characters.

OemStr: Location where translated string is to be copied, can be same as *AnsiStr*.

Returns Always –1.

AnsiToOemBuff

Declaration `procedure AnsiToOemBuff(AnsiStr, OemStr: PChar; Length: Integer);`

Translates *AnsiStr* to OEM-defined character set.

Parameters *AnsiStr*: Buffer of ANSI characters.

OemStr: Location where translated string is to be copied, can be same a *AnsiStr*.

Length: Size of *AnsiStr*, if zero, the length is 64K.

AnsiUpper

Declaration `function AnsiUpper(Str: PChar): PChar;`

Uses language driver to convert *Str* to uppercase.

Parameters *Str*: String (null-terminated) or a single character (in low-order byte).

Returns Converted string or character.

AnsiUpperBuff

Declaration `function AnsiUpperBuff(Str: PChar; Length: Word): Word;`

Uses language driver to convert *Str* to uppercase.

Parameters *Str*: Buffer of characters.

Length: Size of *Str*; if zero length is 64K (65,536).

Returns Length of the converted string.

AnyPopup

Declaration `function AnyPopup: Bool;`

Determines whether a popup window exists on the screen.

Returns Non-zero if popup window exists; zero if not.

AppendMenu

Declaration `function AppendMenu(Menu: HMenu; Flags, IDNewItem: Word; NewItem: PChar): Bool;`

Appends a new item, whose state is specified by *Flags*, to the end of a menu.

Parameters *Menu*: Menu to be changed.

Flags: One or a combination of the following MF constants: *mf_Bitmap, mf_Checked, mf_Disabled, mf_Enabled, mf_Grayed, mf_MenuBarBreak, mf_MenuBreak, mf_OwnerDraw, mf_Popup, mf_Separator, mf_String, mf_Unchecked*. See "(MF_) Menu flags" in Chapter 1.

IDNewItem: Command ID or menu handle if popup menu.

NewItem: New menu string, or in the case of a using a bitmap as an item, the low order word holds a handle to the bitmap.

Returns Non-zero if successful; zero if not.

See also *DrawMenuBar, SetMenuItemBitmaps, wm_DrawItem, wm_MeasureItem.*

Arc

Declaration `function Arc(DC: HDC; X1, Y1, X2, Y2, X3, Y3, X4, Y4: Integer): Bool;`

Draws an elliptical arc centered in the bounding rectangle.

Parameters *DC*: Device context.

X1, Y1: Upper-left corner of bounding rectangle.

X2, Y2: Lower-right corner of bounding rectangle.

X3, Y3: Arc's starting point.

X4, Y4: Arc's end point.

Returns Non-zero if the arc is drawn; zero if not.

See Also The bounding rectangle must not be taller or wider than 32,767 units.

ArrangeIconicWindows

Declaration `function ArrangeIconicWindows(Wnd: HWnd): Word;`

Arranges icons in an MDI client window or icons on the desktop window.

Parameters *Wnd*: Parent window identifier.

Returns Height of one row of icons; zero if no icons.

See also *GetDesktopWindow*.

BeginDeferWindowPos

Declaration `function BeginDeferWindowPos(NumWindows: Integer): THandle;`

Allocates memory for multiple window-position data structure.

Parameters *NumWindows*: Initial number of windows whose position information is to be stored.

Returns Window-position structure identifier.

See also *DeferWindowPos*, *EndDeferWindowPos*.

BeginPaint

Declaration `function BeginPaint(Wnd: HWnd; var Paint: TPaintStruct): HDC;`

Prepares a window for painting in response to a *wm_Paint* message. Fills *Paint* with information for painting.

Parameters *Wnd*: Window to be repainted.

Paint: *TPaintStruct* to receive painting information.

Returns Device context identifier.

See also *EndPaint*, *InvalidateRect*, *InvalidateRgn*, *wm_EraseBkgnd*, *wm_Paint*.

BitBlt

Declaration `function BitBlt(DestDC: HDC; X, Y, Width, Height: Integer; SrcDC: HDC; XSrc, YSrc: Integer; Rop: Longint): Bool;`

Copies a bitmap from *SrcDC* to *DestDC* performing the specified raster operation.

Parameters *DestDC*: Device context to receive the bitmap.

X, Y: Upper-left corner of the destination rectangle.

Width: Width of the destination rectangle and source bitmap.

Height: Height of the destination rectangle and source bitmap.

SrcDC: Device context to copy bitmap from, or zero for raster op on *DestDC* only.

XSrc, YSrc: Upper-left corner of *SrcDC*.

Rop: One of the ternary raster operation constants: *Blackness, DSTInvert, MergeCopy, MergePaint, NotSrcCopy, NotSrcErase, PatCopy, PatInvert, PatPaint, SrcAnd, SrcCopy, SrcErase, SrcInvert, SrcPaint, Whiteness. SrcCopy* performs a simple copy from source to destination.

See "Ternary raster operations" in Chapter 1.

Returns Non-zero if bitmap is drawn; zero if not.

BringWindowToTop

Declaration `procedure BringWindowToTop(Wnd: HWnd);`

Activates and brings *Wnd* to top of a stack of overlapping windows.

Parameters *Wnd*: Pop-up or child window.

BuildCommDCB

Declaration `function BuildCommDCB(Def: PChar; var DCB: TDCB): Integer;`

Translates *Def* into appropriate device-control block codes which are copied into *DCB*.

Parameters *Def*: DOS MODE command-line (null-terminated) of device-control information.

DCB: Receiving *TDCB* structure.

Returns Zero if *Def* is translated; negative if not.

See also *SetCommState*.

CallMsgFilter

Declaration

```
function CallMsgFilter(var Msg: TMsg; Code: Integer): Bool;
```

Passes *Msg* to the current message filter function.

Parameters *Msg*: *TMsg* containing message to be filtered.

Code: Filter function code.

Returns Zero if message should be processed; non-zero if not.

See also *SetWindowsHook*.

CallWindowProc

Declaration

```
function CallWindowProc(PrevWndProc: TFarProc; Wnd: HWnd; Msg, wParam:
Word; lParam: Longint): Longint;
```

Calls and passes message information to *PrevWndProc*. Enables window subclassing by allowing messages to be intercepted before being passed to the window function of the class.

Parameters *PrevWndProc*: Procedure-instance address of the previous window function.

Wnd: Window that receives the message.

Msg: Message identifier.

wParam: Additional message-dependent information.

lParam: Additional message-dependent information.

Returns Value from call to *PrevWndProc*.

See also *SetWindowLong*.

Catch

Declaration `function Catch(var CatchBuf: TCatchBuf): Integer;`

Copies the state of all systems registers and the instruction pointer into *CatchBuf*.

Parameters *CatchBuf*: *TCatchBuf* to copy execution environment into.

Returns Zero if the environment is copied.

See also *Throw*.

ChangeClipboardChain

Declaration `function ChangeClipboardChain(Wnd, WndNext: HWnd): Bool;`

Removes *Wnd* from the chain of clipboard viewers and replaces it with *WndNext*.

Parameters *Wnd*: Window to be removed from chain.

WndNext: Window that follows *Wnd* in the clipboard-viewer chain.

Returns Non-zero if the window is found and removed.

See also *SetClipboardViewer*, *wm_ChangeCBChain*.

CheckDlgButton

Declaration `procedure CheckDlgButton(Dlg: HWnd; IDButton: Integer; Check: Word);`

Places or removes a checkmark from a button control, or changes state of a three-button control.

Parameters *Dlg*: Dialog box that contains the button.

IDButton: Button control to be modified.

Check: Removed(0), checked(1), grayed(2).

CheckMenuItem

Declaration `function CheckMenuItem(Menu: HMenu; IDCheckItem, Check: Word): Bool;`

Places or removes a checkmark from menu items in a popup menu.

Parameters *Menu*: Popup menu.

IDCheckItem: Menu item to be checked.

Check: How item should be checked and how its position is specified. Can be a combination of *mf_ByCommand* or *mf_ByPosition* and *mf_Checked* or *mf_Unchecked*. See "(mf_) Menu flags" in Chapter 1.

Returns Previous state of item; –1 if menu item does not exist.

CheckRadioButton

Declaration `procedure CheckRadioButton(Dlg: HWnd; IDFirstButton, IDLastButton, IDCheckButton: Integer);`

Checks *IDCheckButton* and removes the checkmark from the group of radio buttons specified by *IDFirstButton* and *IDLastButton*.

Parameters *Dlg*: Dialog box.

IDFirstButton: ID of first radio button in group.

IDLastButton: ID of last radio button in group.

IDCheckButton: ID of radio button to check.

ChildWindowFromPoint

Declaration `function ChildWindowFromPoint(WndParent: HWnd; APoint: TPoint): HWnd;`

Determines which child window owned by *WndParent* contains *APoint*.

Parameters *WndParent*: Parent window

APoint: *TPoint* structure of client coordinates to be tested

Returns Child window that contains the point; zero if point lies outside the parent window; *WndParent* if point not contained in within any child window.

Chord

Declaration `function Chord(DC: HDC; X1, Y1, X2, Y2, X3, Y3, X4, Y4: Integer): Bool;`

Draws a chord bounded by the intersection of an ellipse which is centered in the bounding rectangle and a line segment.

Parameters *DC*: Device context.

X1, Y1: Upper-left corner of bounding rectangle.

X2, Y2: Lower-right corner of bounding rectangle.

X3, Y3: One end of the line segment.

X4, Y4: One end of the line segment.

Returns Non-zero if the chord is drawn; zero if not.

ClearCommBreak

Declaration `function ClearCommBreak(Cid: Integer): Integer;`

Restores character transmission and places the line in a non-break state.

Parameters *Cid*: Communication device to be restored.

Returns Zero if successful; negative if *Cid* is not a valid device.

See also *OpenComm*.

ClientToScreen

Declaration `procedure ClientToScreen(Wnd: HWnd; var Point: TPoint);`

Converts client coordinates in *APoint* to screen coordinates.

Parameters *Wnd*: Window containing client area.

APoint: *TPoint* containing client coordinates.

ClipCursor

Declaration `procedure ClipCursor(Rect: LPRect);`

Confines cursor to *Rect*. If *Rect* is **nil** the cursor is unbounded.

Parameters *Rect*: Bounding *TRect* in screen coordinates.

See also *SetCursorPos*.

CloseClipboard

Declaration `function CloseClipboard: Bool;`

Closes the clipboard to allow access by other applications.

Returns Non-zero if clipboard is closed; zero if not.

CloseComm

Declaration `function CloseComm(Cid: Integer): Integer;`

Closes *Cid*, flushing the output queue. Any memory used for transmit and receive queues is freed.

Parameters *Cid*: Communications device.

Returns Zero if device is closed; negative if error.

See also *OpenComm*.

CloseMetaFile

Declaration `function CloseMetaFile(DC: THandle): THandle;`

Closes *DC* and creates a metafile handle that can be used to play the metafile.

Parameters *DC*: Metafile device context.

Returns Metafile identifier if successful; 0 if not.

See also *PlayMetaFile.*

CloseSound

Declaration `procedure CloseSound;`

Flushes all voice queues, frees any allocated buffers and closes access to the play device.

CloseWindow

Declaration `procedure CloseWindow(Wnd: HWnd);`

Minimizes *Wnd*. Icons for overlapped windows are moved into the icon area of the screen.

Parameters *Wnd*: Window to be minimized.

CombineRgn

Declaration `function CombineRgn(DestRgn, SrcRgn1, SrcRgn2: HRgn; CombineMode: Integer): Integer;`

Combines regions *SrcRgn1* with *SrcRgn2* and places result in *DestRgn*. *CombineMode* specifies the method with which the regions are combined.

Parameters *DestRgn*: Region to be replaced with new region.

SrcRgn1: An existing region.

SrcRgn2: An existing region.

CombineMode: One of the constants *rgn_And, rgn_Copy, rgn_Diff, rgn_Or, rgn_Xor*. See "(rgn_) Combine region flags" in Chapter 1.

Returns One of the constants *ComplexRegion, Error, NullRegion, SimpleRegion*. See "Region flags" in Chapter 1.

CopyMetaFile

Declaration `function CopyMetaFile(SrcMetaFile: THandle; FileName: PChar): THandle;`

Copies *SrcMetaFile* to the file *FileName*.

Parameters *SrcMetaFile*: Source metafile.

FileName: Metafile name (null-terminated) or 0 to copy to a memory metafile.

Returns New metafile identifier.

CopyRect

Declaration `procedure CopyRect(var DestRect, SourceRect: TRect);`

Copies *SourceRect* to *DestRect*.

Parameters *DestRect*: *TRect* structure.

SourceRect: *TRect* structure.

CountClipboardFormats

Declaration `function CountClipboardFormats: Integer;`

Counts number of formats the clipboard can render.

Returns Number of data formats in clipboard.

CountVoiceNotes

Declaration `function CountVoiceNotes(Voice: Integer): Integer;`

Counts the number of notes in *Voice*.

Parameters *Voice*: Voice queue.

Returns Number of notes.

See also *SetVoiceNote*.

CreateBitmap

Declaration `function CreateBitmap(Width, Height: Integer; Planes, BitCount: Byte; Bits: Pointer): HBitmap;`

Creates a device-dependent memory bitmap.

Parameters *Width*: Width (in pixels) of the bitmap.

Height: Height (in pixels) of the bitmap.

Planes: Number of color planes in the bitmap.

BitCount: Number of color bits per display pixel.

Bits: Short-integer array containing initial bitmap values; if **nil** the new bitmap is left uninitialized.

Returns Bitmap identifier if successful; 0 if not.

See also *BitBlt*, *SelectObject*.

CreateBitmapIndirect

Declaration `function CreateBitmapIndirect(var Bitmap: TBitmap): HBitmap;`

Creates a bitmap defined by *Bitmap*.

Parameters *Bitmap*: *TBitmap* structure.

Returns Bitmap identifier if successful; 0 if not.

See also *BitBlt*.

CreateBrushIndirect

Declaration `function CreateBrushIndirect(var LogBrush: TLogBrush): HBrush;`

Creates a logical brush defined by *LogBrush*.

Parameters *LogBrush*: *TLogBrush* structure.

Returns Logical brush identifier if successful; 0 if not.

CreateCaret

Declaration `procedure CreateCaret(Wnd: HWnd; ABitmap: HBitmap; Width, Height: Integer);`

Creates a new shape for the system caret.

Parameters *Wnd*: Owning window of the new caret.

ABitmap: Bitmap that defines the caret; if 0, caret is black; if, 1 caret is gray.

Width: Caret width (in logical units).

Height: Caret height (in logical units).

See also *CreateBitmap, CreateDIBitmap, GetSystemMetrics, LoadBitmap.*

CreateCompatibleBitmap

Declaration `function CreateCompatibleBitmap(DC: HDC; Width, Height: Integer): HBitmap;`

Creates a bitmap that is compatible with *DC*.

Parameters *DC*: Device context.

Width: Bitmap width (in bits).

Height: Bitmap height (in bits).

Returns Bitmap identifier if successful; 0 if not.

CreateCompatibleDC

Declaration `function CreateCompatibleDC(DC: HDC): HDC;`

Creates a memory device context that is compatible with *DC*.

Parameters *DC*: Device context; if 0 a memory device context is created.

Returns New memory device context if successful; 0 if not.

See also *DeleteDC, GetDeviceCaps.*

CreateCursor

Declaration `function CreateCursor(Instance: THandle; Xhotspot, Yhotspot, Width, Height: Integer; ANDBitPlane, XORBitPlane: Pointer): HCursor;`

Creates a cursor.

Parameters *Instance*: Module instance creating the cursor.

Xhotspot, Yhotspot: Position of cursor hotspot.

Width: Cursor width (in pixels).

Height: Cursor height (in pixels).

ANDBitPlane: Array of bytes containing **AND** mask.

XORBitPlane: Array of bytes containing **XOR** mask.

Returns Cursor identifier if successful; 0 if not.

CreateDC

Declaration `function CreateDC(DriverName, DeviceName, Output: PChar; InitData: Pointer): HDC;`

Creates a device context for *DriverName* device.

Parameters *DriverName*: DOS filename (without extension and null-terminated) of the device driver.

DeviceName: Name of specific device to be supported (null-terminated).

Output: Output DOS file or device name (null-terminated).

InitData: *TDevMode* structure containing device-specific initialization data.

Returns Device context identifier if successful; 0 if not.

CreateDialog

Declaration `function (Instance: THandle; TemplateName: PChar; WndParent: HWnd; DialogFunc: TFarProc): HWnd;`

Creates a modeless dialog box defined by *TemplateName* dialog box resource.

Parameters *Instance*: Module instance whose executable file contains the dialog box resource.

TemplateName: Dialog box resource name (null-terminated).

WndParent: Parent window of the dialog box.

DialogFunc: Dialog function procedure-instance address or **nil** if class defined.

Returns Dialog box window handle if successful; 0 if not.

See also *DefDlgProc, MakeProcInstance, wm_InitDialog.*

CreateDialogIndirect

Declaration

```
function CreateDialogIndirect(Instance: THandle; DialogTemplate: PChar;
Parent: HWnd; DialogFunc: TFarProc): HWnd;
```

Creates a modeless dialog box defined by *DialogTemplate.*

Parameters *Instance*: Module instance.

DialogTemplate: *TDlgTemplate* structure containing dialog box template.

WndParent: Owning window of the dialog box.

DialogFunc: Dialog callback function procedure-instance address.

Returns Dialog box window handle if successful; 0 if not.

See also *DefDlgProc, MakeProcInstance, wm_InitDialog.*

CreateDialogIndirectParam

Declaration

```
function CreateDialogIndirectParam(Instance: THandle; var DialogTemplate;
WndParent: HWnd; DialogFunc: TFarProc; InitParam: Longint): HWnd;
```

Creates a modeless dialog box defined by *DialogTemplate.* It differs from *CreateDialogIndirect* in that it allows you to pass a parameter, *InitParam*, to the callback function.

Parameters *Instance*: Module instance.

DialogTemplate: *TDlgTemplate* structure containing dialog box template.

WndParent: Owning window of the dialog box.

DialogFunc: Dialog function procedure-instance address or **nil** if class defined.

InitParam: Value passed to dialog function (*lParam* of *wm_InitDialog* message) when the dialog box is created.

Returns Dialog box window handle if successful; 0 if not.

See also *DefDlgProc, MakeProcInstance, wm_InitDialog.*

CreateDialogParam

Declaration

```
function CreateDialogParam(Instance: THandle; TemplateName: PChar;
WndParent: HWnd; DialogFunc: TFarProc; InitParam: Longint): HWnd;
```

Creates a modeless dialog box defined by *TemplateName.*

Parameters *Instance*: Module instance whose executable file contains the dialog box template.

TemplateName: Dialog box template name (null-terminated).

Parent: Owning window of the dialog box.

DialogFunc: Dialog function procedure-instance address or **nil** if class defined.

InitParam: Value passed to dialog function (*lParam* of *wm_InitDialog* message) when the dialog box is created.

Returns Dialog box window handle if successful; 0 if not.

See also *DefDlgProc, MakeProcInstance, wm_InitDialog.*

CreateDIBitmap

Declaration

```
function CreateDIBitmap(DC: HDC; var InfoHeader: TBitmapInfoHeader;
Usage: Longint; InitBits: PChar; var InitInfo: TBitmapInfo; Usage: Word):
HBitmap;
```

Creates a device-specific memory bitmap from a device-independent bitmap described by *InfoHeader* and *InitInfo.*

Parameters *DC*: Device context.

InfoHeader: *TBitmapInfoHeader* which describes bitmap size and format.

Usage: If *cbm_Init* the bitmap is initialized according to *InitBits* and *InitInfo.*

InitBits: Byte array containing initial bitmap values, format dependent on *biBitCount* field of *InitInfo*.

InitInfo: *TBitmapInfo* structure which describes dimensions and color format.

Usage: One of the constants *DIB_RGB_Colors*, or *DIB_Pal_Colors*. See "(DIB_) Color table identifiers" in Chapter 1.

Returns Bitmap identifier if successful; 0 if not.

CreateDIBPatternBrush

Declaration `function CreateDIBPatternBrush(PackedDIB: THandle; Usage: Word): HBrush;`

Creates a logical brush from the device-independent bitmap defined by *PackedDIB*.

Parameters *PackedDIB*: global memory containing *TBitmapInfo* structure plus array of pixels.

Usage: *DIB_RGB_Colors*, *DIB_Pal_Colors*. See "(DIB_) Color table identifiers" in Chapter 1.

Returns logical brush identifier if successful; 0 if not.

See also *GetDeviceCaps*.

CreateDiscardableBitmap

Declaration `function CreateDiscardableBitmap(DC: HDC; Width, Height: Integer): HBitmap;`

Creates a discardable bitmap compatible with *DC*.

Parameters *DC*: Device context.

Width: Bitmap width (in bits).

Height: Bitmap height (in bits).

Returns Bitmap identifier if successful; 0 if not.

CreateEllipticRgn

Declaration `function CreateEllipticRgn(X1, Y1, X2, Y2: Integer): HRgn;`

Creates an elliptical region bounded by the specified rectangle.

Parameters *X1, Y1*: Upper-left corner of bounding rectangle.

X2, Y2: Lower-right corner of bounding rectangle.

Returns New region identifier if successful; 0 if not.

CreateEllipticRgnIndirect

Declaration `function CreateEllipticRgnIndirect(var Rect: TRect): HRgn;`

Creates an elliptical region bounded by the rectangle specified in *ARect*.

Parameters *ARect*: *TRect* containing upper-left and lower-right corners of the bounding rectangle.

Returns New region identifier if successful; 0 if not.

CreateFont

Declaration
```
function CreateFont(Height, Width, Escapement, Orientation, Weight:
Integer; Italic, Underline, StrikeOut, CharSet, OutputPrecision,
ClipPrecision, Quality, PitchAndFamily: Byte; FaceName: PChar): HFont;
```

Creates a logical font selected from GDI's pool of physical fonts according to specified characteristics.

Parameters *Height*: Font height (in logical units).

Width: Average font width (in logical units).

Escapement: Line angle (in tenths of degrees).

Orientation: Character baseline angle (in tenths of degrees).

Weight: Font weight (0 – 1000). Or use *fw_* constants such as *fw_Normal* of *fw_Bold*. See "(fw_) Font weight flags" in Chapter 1.

Italic: Font is italic.

Underline: Font is underlined.

StrikeOut: Font is struck out.

CharSet: One of the constants *ANSI_CharSet, OEM_CharSet, Symbol_CharSet.*

OutputPrecision: One of the constants *Out_Character_Precis, Out_Default_Precis, Out_String_Precis, Out_Stroke_Precis.*

ClipPrecision: One of the constants *Clip_Character_Precis, Clip_Default_Precis, Clip_Stroke_Precis.*

Quality: One of the constants *Default_Quality, Draft_Quality, Proof_Quality.* See "Font output quality flags" in Chapter 1.

PitchAndFamily: One of the constants *Default_Pitch, Fixed_Pitch,* or *Variable_Pitch* combined with one of *ff_Decorative, ff_DontCare, ff_Modern, ff_Roman, ff_Script,* or *ff_Swiss.* See "(ff_) Font family flags" in Chapter 1.

Facename: Font typeface name (null-terminated).

Returns Logical font identifier if successful; 0 if not.

See also *EnumFonts.*

CreateFontIndirect

Declaration

```
function CreateFontIndirect(var LogFont: TLogFont): HFont;
```

Creates a logical font selected from GDI's pool of physical fonts according to the characteristics in *ALogFont.*

Parameters *ALogFont*: *TLogFont* structure.

Returns Logical font identifier if successful; 0 if not.

CreateHatchBrush

Declaration

```
function CreateHatchBrush(Index: Integer; Color: TColorRef): HBrush;
```

Creates a logical brush with the specified hatch style.

Parameters *Index*: One of the constants: *hs_BDiagonal, hs_Cross, hs_DiagCross, hs_FDiagonal, hs_Horizontal,* or *hs_Vertical.* See "(hs_) Hatch styles" in Chapter 1.

Color: *TColorRef* that specifies the color of the hatches.

Returns Logical brush identifier if successful; 0 if not.

CreateIC

Declaration `function CreateIC(DriverName, DeviceName, Output, InitData: PChar): HDC;`

Creates an information context for the device.

Parameters *DriverName*: Device driver DOS filename (no extension and null-terminated).

DeviceName: Specific device name (null-terminated).

Output: Output DOS file or device name (null-terminated).

InitData: Device-specific initialization data; **nil** for default initialization.

Returns Information context identifier if successful; 0 if not.

CreateIcon

Declaration `function CreateIcon(Instance: THandle; Width, Height: Integer; Planes, BitsPixel: Byte; ANDbits, XORbits: Pointer): HIcon;`

Creates an icon.

Parameters *Instance*: Module instance creating the icon.

Width: Icon width (in pixels).

Height: Icon height (in pixels).

Planes: Number of planes in **XOR** mask.

BitsPixel: Number of bits per pixel in **XOR** mask.

ANDbits: Byte array containing monochrome **AND** mask of the icon.

XORbits: Byte array containing **XOR** mask.

Returns Icon identifier if successful; 0 if not of the icon.

CreateMenu

Declaration `function CreateMenu: HMenu;`

Creates an initially empty menu.

Returns Menu identifier if successful; 0 if not.

See also *AppendMenu, InsertMenu.*

CreateMetaFile

Declaration `function CreateMetaFile(FileName: PChar): THandle;`

Creates a metafile device context.

Parameters *FileName*: Metafile name (null-terminated) or **nil** to specify a memory metafile.

Returns Metafile device context identifier if successful; 0 if not.

CreatePalette

Declaration `function CreatePalette(var LogPalette: TLogPalette): HPalette;`

Creates a logical color palette.

Parameters *LogPalette*: *TLogPalette* containing color information about the logical palette.

Returns Logical palette identifier if successful; 0 if not.

CreatePatternBrush

Declaration `function CreatePatternBrush(Bitmap: HBitmap): HBrush;`

Creates a logical brush with *Bitmap* pattern.

Parameters *Bitmap*: *HBitmap* bitmap identifier.

Returns Logical brush identifier if successful; 0 if not.

See also *CreateBitmap, CreateBitmapIndirect, LoadBitmap, CreateCompatibleBitmap, DeleteObject, GetDeviceCaps.*

CreatePen

Declaration `function CreatePen(PenStyle, Width: Integer; Color: TColorRef): HPen;`

Creates a logical pen.

Parameters *PenStyle*: One of the constants *ps_Solid, ps_Dash, ps_Dot, ps_DashDot, ps_DashDotDot, ps_Null,* or *ps_InsideFrame.* See "(ps_) Pen styles" in Chapter 1.

Width: Pen width (in logical units).

Color: Pen *TColorRef.*

Returns Logical pen identifier if successful; 0 if not.

CreatePenIndirect

Declaration `function CreatePenIndirect(var LogPen: TLogPen): HPen;`

Creates a logical pen defined by *LogPen.*

Parameters *LogPen*: *TLogPen* structure.

Returns Logical pen identifier if successful; 0 if not.

CreatePolygonRgn

Declaration `function CreatePolygonRgn(var Points; Count, PolyFillMode: Integer): HRgn;`

Creates a polygon region.

Parameters *Points*: *TPoint* array containing vertices of the polygon.

Count: Number of points in *Points*.

PolyFillMode: Mode for filling the region; use one of the constants *Alternate* or *Winding*. See "PolyFill modes" in Chapter 1.

Returns New region identifier if successful; 0 if not.

See also *SetPolyFillMode.*

CreatePolyPolygonRgn

Declaration

```
function CreatePolyPolygonRgn(var Points; var PolyCounts; Counts,
PolyFillMode: Integer): HRgn;
```

Creates a region consisting of a series of possibly overlapping closed polygons.

Parameters *Points*: *TPoint* array containing vertices of the polygons.

PolyCounts: Integer array where each corresponding element specifies the number of points for each polygon in *Points*.

Count: Size of *PolyCounts*.

PolyFillMode: *Alternate*, or *Winding*. See "PolyFill modes" in Chapter 1, "Windows styles and constants."

Returns Region identifier if successful; 0 if not.

CreatePopupMenu

Declaration

```
function CreatePopupMenu: HMenu;
```

Creates an empty popup menu.

Returns Menu identifier if successful; 0 if not.

See also *AppendMenu, InsertMenu, TrackPopupMenu.*

CreateRectRgn

Declaration

```
function CreateRectRgn(X1, Y1, X2, Y2: Integer): HRgn;
```

Creates a rectangular region bounded by the specified rectangle.

Parameters *X1, Y1*: Upper-left corner of bounding rectangle.

X2, Y2: Lower-right corner of bounding rectangle.

Returns Region identifier if successful; 0 if not.

CreateRectRgnIndirect

Declaration `function CreateRectRgnIndirect(var Rect: TRect): HRgn;`

Creates a rectangular region bounded by *ARect*.

Parameters *ARect*: *TRect* containing upper-left and lower-right corners of the region.

CreateRoundRectRgn

Declaration `function CreateRoundRectRgn(X1, Y1, X2, Y2, X3, Y3: Integer): HRgn;`

Creates a rectangular region with rounded corners bounded by the specified region.

Parameters *X1, Y1*: Upper-left corner of region.

X2, Y2: Lower-right corner of region.

X3: Ellipse width for rounded corners.

Y3: Ellipse height for rounded corners.

Returns Region identifier if successful; 0 if not.

CreateSolidBrush

Declaration `function CreateSolidBrush(Color: TColorRef): HBrush;`

Creates a logical brush.

Parameters *Color*: Brush *TColorRef*.

Returns Logical brush identifier if successful; 0 if not.

CreateWindow

Declaration
```
function CreateWindow(ClassName, WindowName: PChar; Style: Longint; X, Y,
Width, Height: Integer; WndParent: HWnd; Menu: HMenu; Instance: THandle;
Param: Pointer): HWnd;
```

Creates an overlapped, pop-up, or child window.

Parameters *ClassName*: Window class name (null-terminated) or a predefined control-class name.

WindowName: Window caption or name (null-terminated).

Style: One or a combination of window or control style constants, including *ds_*, *ws_*, *bs_*, *cbs_*, *es_*, *lbs_*, *sbs_*, or *ss_* constants. See Chapter 1.

X, *Y*: Initial window position or *cw_UseDefault*. See "(cw_) Create window default code" in Chapter 1.

Width: Initial window width (in device units).

Height: Initial window height (in device units).

WndParent: Owner window.

Menu: Menu or child-window identifier.

Instance: Associated module instance.

Param: Value passed in *TCreateStruct* in *lParam* of the *wm_Create* message, must be a pointer to a *TClientCreateStruct* for MDI client window creation.

Returns Window identifier if successful; 0 if not.

See also *RegisterClass*, *wm_Create*, *wm_GetMinMaxInfo*, *wm_NCCreate*.

CreateWindowEx

Declaration

```
function CreateWindowEx(ExStyle: Longint; ClassName, WindowName: PChar;
Style: Longint; X, Y, Width, Height: Integer; WndParent: HWnd; Menu:
HMenu; Instance: THandle; Param: Pointer): HWnd;
```

Creates an overlapped, pop-up, or child window with an extended style.

Parameters

ExStyle: One of these extended window styles: *ws_ex_DlgModalFrame* or *ws_ex_NoParentNotify*. See "(ws_ex_) Extended window styles" in Chapter 1.

ClassName: Window class name (null-terminated) or a predefined control-class name.

WindowName: Window caption or name (null-terminated).

Style: One or a combination of window or control style constants, including *ds_*, *ws_*, *bs_*, *cbs_*, *es_*, *lbs_*, *sbs_*, or *ss_* constants. See Chapter 1.

X, *Y*: Initial window position or *cw_UseDefault*. See "(cw_) Create window default code" in Chapter 1.

Width: Initial window width (in device units).

Height: Initial window height (in device units).

WndParent: Owner window.

Menu: Menu or child-window identifier.

Instance: Associated module instance.

Param: Value passed in *TCreateStruct* in *lParam* of the *wm_Create* message, must be a pointer to a *TClientCreateStruct* for MDI client window creation.

Returns Window identifier if successful; 0 if not.

See also *CreateWindow*, *wm_ParentNotify*.

DebugBreak

Declaration

```
procedure DebugBreak;
```

Forces a break to the debugger.

DefDlgProc

Declaration

```
function DefDlgProc(Dlg: HWnd; Msg, wParam: Word; lParam: Longint):
Longint;
```

Provides default processing for dialogs with a private window class.

Parameters *Dlg*: Dialog box identifier.

Msg: Message number.

wParam: message-dependent information.

lParam: message-dependent information.

Returns Result of the message processing.

DeferWindowPos

Declaration

```
function DeferWindowPos(WinPosInfo: THandle; Wnd, WndInsertAfter: HWnd;
X, Y, cX, cY: Integer; Flags: Word): THandle;
```

Updates *WinPosInfo* for the window identified by *Wnd*.

Parameters *WinPosInfo*: Multiple window-position data structure identifier.

Wnd: Window to store update information about.

WndInsertAfter: Window to insert *Wnd* after.

X, *Y*: Windows upper-left corner position.

cX: Window's new width.

cY: Window's new height.

Flags: One of *swp_DrawFrame*, *swp_HideWindow*, *swp_NoActivate*, *swp_NoMove*, *swp_NoSize*, *swp_NoRedraw*, *swp_NoZOrder*, or *swp_ShowWindow*. See "(swp_) Set window position flags" in 1, "Windows styles and constants."

Returns Updated multiple window-position data structure.

See also *BeginDeferWindowPos*, *EndDeferWindowPos*

DefFrameProc

Declaration

```
function DefFrameProc(Wnd, MDIClient: HWnd; Msg, wParam: Word; lParam:
Longint): Longint;
```

Provides default message processing for MDI frame windows.

Parameters *Wnd*: MDI frame window.

MDIClient: MDI client window.

Msg: Message number.

wParam: Message dependent information.

lParam: Message dependent information.

Returns Result of message processing.

DefHookProc

Declaration

```
function DefHookProc(Code: Integer; wParam: Word; lParam: Longint;
NextHook: TFarProc): Longint;
```

Calls next function in chain of hook (message filter) functions.

Parameters *Code*: Determines how message is processed.

wParam: Message word parameter.

lParam: Message long parameter.

NextHook: *TFarProc* to next hook function.

Returns A value dependent upon code.

See also *SetWindowsHook, UnhookWindowsHook.*

DefMDIChildProc

Declaration

```
function DefMDIChildProc(Wnd: HWnd; Msg, wParam: Word; lParam: Longint):
Longint;
```

Provides default message processing for MDI child windows.

Parameters *Wnd*: MDI child window.

Msg: Message number.

wParam: Message-dependent information.

lParam: Message-dependent information.

Returns Result of the message processing.

DefWindowProc

Declaration

```
function DefWindowProc(Wnd: HWnd; Msg, wParam: Word; lParam: Longint):
Longint;
```

Provides default message processing for messages not explicitly handled by the application.

Parameters *Wnd*: Window identifier.

Msg: Message number.

wParam: Message-dependent information.

lParam: Message-dependent information.

Returns Result of the message processing.

DeleteAtom

Declaration `function DeleteAtom(AnAtom: TAtom): TAtom;`

Deletes an atom. If the atom's reference count is zero the associated string will be removed from the atom table.

Parameters *AnAtom*: Atom identifier.

Returns 0 if successful; Atom if not.

DeleteDC

Declaration `function DeleteDC(DC: HDC): Bool;`

Deletes the device context. Notifies the device and releases all storage and systems resources if *DC* is the last device context for the device.

Parameters *DC*: Device context identifier.

Returns Non-zero if successful; zero if not.

DeleteMenu

Declaration `function DeleteMenu(Menu: HMenu; Position, Flags: Word): Bool;`

Deletes an item from *Menu*. If the item is a popup its handle is destroyed and memory freed.

Parameters *Menu*: Menu identifier.

Position: Position or command ID.

Flags: One of the menu constants: *mf_ByPosition, mf_ByCommand*. See "(mf_) Menu flags" in Chapter 1.

Returns Non-zero if successful; zero if not.

DeleteMetaFile

Declaration `function DeleteMetaFile(MF: THandle): Bool;`

Invalidates a metafile handle and frees its associated system resources. The metafile itself is not deleted.

Parameters *MF*: Metafile identifier.

Returns Non-zero if successful; zero if *MF* is not a valid handle.

See also *GetMetaFile*.

DeleteObject

Declaration `function DeleteObject(Handle: THandle): Bool;`

Deletes *Handle* from memory and frees all associated system resources.

Parameters *Handle*: Handle to a logical pen, brush, font, bitmap, region, or palette.

Returns Non-zero if deleted; zero if *Handle* is not valid or currently selected into a device context.

DestroyCaret

Declaration `procedure DestroyCaret;`

Destroys the current caret, frees it from the owning window and removes it from the screen (if visible).

DestroyCursor

Declaration `function DestroyCursor(Cursor: HCursor): Bool;`

Destroys *Cursor* and frees associated memory.

Parameters *Cursor*: Cursor identifier.

Returns Non-zero if successful; zero if not.

See also *CreateCursor*.

DestroyIcon

Declaration `function DestroyIcon(Icon: HIcon): Bool;`

Destroys *Icon* and frees associated memory.

Parameters *Icon*: Icon identifier.

Returns Non-zero if successful; zero if not.

See also *CreateIcon*.

DestroyMenu

Declaration `function DestroyMenu(Menu: HMenu): Bool;`

Destroys *Menu* and frees associated memory.

Parameters *Menu*: Menu identifier.

Returns Non-zero if successful; zero if not.

DestroyWindow

Declaration `function DestroyWindow(Wnd: HWnd): Bool;`

Destroys a window or a modeless dialog box and any associated child windows.

Parameters *Wnd*: Window identifier.

Returns Non-zero if successful; zero if not.

See also *CreateDialog, wm_Destroy, wm_NCDestroy.*

DialogBox

Declaration `function DialogBox(Instance: THandle; TemplateName: PChar; WndParent: HWnd; DialogFunc: TFarProc): Integer;`

Creates a modal dialog box defined by *TemplateName* and sends *wm_InitDialog* before displaying the dialog.

Parameters *Instance*: Module instance whose executable file contains the dialog box template.

TemplateName: Dialog box template name (null-terminated).

WndParent: Owning window.

DialogFunc: Dialog callback function procedure-instance address.

Returns *EndDialog nResult* parameter; –1 if dialog could not be created.

DialogBoxIndirect

Declaration `function DialogBoxIndirect(Instance, DialogTemplate: THandle; WndParent: HWnd; DialogFunc: TFarProc): Integer;`

Creates a modal dialog box defined by *DialogTemplate* and sends *wm_InitDialog* before displaying the dialog.

Parameters *Instance*: Module instance whose executable file contains the dialog box template.

DialogTemplate: Global memory block containing *TDlgTemplate* structure.

WndParent: Owning window.

DialogFunc: Dialog callback function procedure-instance address.

Returns *EndDialog nResult* parameter; –1 if dialog could not be created.

DialogBoxIndirectParam

Declaration `function DialogBoxIndirectParam(Instance, DialogTemplate: THandle; WndParent: HWnd; DialogFunc: TFarProc; InitParam: Longint): Integer;`

Creates a modal dialog box defined by *DialogTemplate* and sends *wm_InitDialog* before displaying the dialog. Also allows you to send an initial parameter to the callback function.

Parameters *Instance*: Module instance whose executable file contains the dialog box template.

DialogTemplate: Global memory block containing *TDlgTemplate* structure.

WndParent: Owning window.

DialogFunc: Dialog function procedure-instance address.

InitParam: Passed in *lParam* of *wm_InitDialog* message.

Returns *EndDialog nResult* parameter; –1 if dialog could not be created.

DialogBoxParam

Declaration

```
function DialogBoxParam(Instance: THandle; TemplateName: PChar; Parent:
HWnd; DialogFunc: TFarProc; InitParam: Longint): Integer;
```

Creates a modal dialog box defined by *TemplateName* and sends *wm_InitDialog* before displaying the dialog. Also allows you to send an initial parameter to the callback function.

Parameters *Instance*: Module instance whose executable file contains the dialog box template.

TemplateName: Dialog box template name (null-terminated).

Parent: Owning window.

DialogFunc: Dialog function procedure-instance address.

InitParam: Passed in *lParam* of *wm_InitDialog* message.

Returns *EndDialog nResult* parameter; –1 if dialog could not be created.

DispatchMessage

Declaration

```
function DispatchMessage(var Msg: TMsg): Longint;
```

Passes the message in *Msg* to the window's window function.

Parameters *Msg*: *TMsg* structure.

Returns Value returned from the window function, generally ignored.

DlgDirList

Declaration `function DlgDirList(Dlg: HWnd; PathSpec: PChar; IDListBox, IDStaticPath: Integer; Filetype: Word): Integer;`

Fills *IDListBox* with a file or directory listing matching the pathname given by *PathSpec*.

Parameters *Dlg*: Dialog box containing *IDListBox*.

PathSpec: Pathname string (null-terminated).

IDListBox: List box control ID.

IDStaticPath: Static-text control ID to display current drive and directory.

Filetype: $0000 (read\write), $0001 (read-only), $0002 (hidden), $0004 (system), $0010 (subdirectories), $0020 archives), $2000 (*lb_Dir*), $4000(drives), $8000 (exclusive).

Returns Non-zero if listing made, zero if invalid search path.

See also *lb_ResetContent*, *lb_Dir*.

DlgDirListComboBox

Declaration `function DlgDirListComboBox(Dlg: HWnd; PathSpec: PChar; IDComboBox, IDStaticPath: Integer; Filetype: Word): Integer;`

Fills *IDComboBox* with a file or directory listing matching the pathname given by *PathSpec*.

Parameters *Dlg*: Dialog box containing *IDComboBox*.

PathSpec: Pathname string (null-terminated).

IDComboBox: Combo box control ID.

IDStaticPath: Static-text control ID to display current drive and directory.

Filetype: $0000 (read\write), $0001 (read-only), $0002 (hidden), $0004 (system), $0010 (subdirectories), $0020 archives), $2000 (*lb_Dir*), $4000(drives), $8000 (exclusive).

Returns Non-zero if listing made, zero if invalid search path.

See also *cb_ResetContent*, *cb_Dir*.

DlgDirSelect

Declaration `function DlgDirSelect(Dlg: HWnd; Str: PChar; IDListBox: Integer): Bool;`

Retrieves current list box selection and fills *Str*.

Parameters *Dlg*: Dialog box containing *IDListBox*.

Str: Pathname buffer.

IDListBox: List box control ID.

Returns Non-zero if current selection is a directory; zero if not.

See also *DlgDirList, lb_GetCurSel, lb_GetText.*

DlgDirSelectComboBox

Declaration `function DlgDirSelectComboBox(Dlg: HWnd; Str: PChar; IDComboBox: Integer): Bool;`

Retrieves current combo box selection, from a simple combo box (*cbs_Simple*) only, and fills *Str*.

Parameters *Dlg*: Dialog box containing *IDComboBox*.

Str: Pathname buffer.

IDComboBox: Combo box control ID.

Returns Non-zero if current selection is a directory; zero if not.

See also *DlgDirListComboBox, cb_GetCurSel, cb_GetText.*

DPtoLP

Declaration `function DPtoLP(DC: HDC; var Points; Count: Integer): Bool;`

Converts device points into logical points.

Parameters *DC*: Device context identifier.

Points: Array of *TPoint* structures.

Count: Number of points in *Points*.

Returns Non-zero if all points are converted; zero if not.

DrawFocusRect

Declaration
```
procedure DrawFocusRect(DC: HDC; var Rect: TRect);
```

Performs **XOR** to draw a focus style rectangle.

Parameters *DC*: Device context identifier.

Rect: *TRect* to be drawn.

DrawIcon

Declaration
```
function DrawIcon(DC: HDC; X, Y: Integer; Icon: HIcon): Bool;
```

Draws an icon.

Parameters *DC*: Device context identifier.

X, Y: Upper-left corner of the icon.

Icon: Icon to be drawn.

Returns Non-zero if successful; zero if not.

DrawMenuBar

Declaration
```
procedure DrawMenuBar(Wnd: HWnd);
```

Redraws a windows menu bar. Used if the menu bar has been changed after the window is created.

Parameters *Wnd*: Window identifier.

DrawText

Declaration `function DrawText(DC: HDC; Str: PChar; Count: Integer; var Rect: TRect; Format: Word): Integer;`

Draws formatted text. The type of formatting is specified by *Format*. Unless otherwise specified by *dt_NoClip* the text is clipped to the bounding rectangle.

Parameters *DC*: Device-context identifier.

Str: String to be drawn. If *Count* is –1 must be null-terminated.

Count: Number of bytes in string.

Rect: Bounding *TRect* of the text.

Format: One or more of the constants: *dt_Bottom, dt_CalcRect, dt_Center, dt_ExpandTabs, dt_ExternalLeading, dt_Left, dt_NoClip, dt_NoPrefix, dt_Right, dt_SingleLine, dt_TabStop, dt_Top, dt_VCenter*, and *dt_WordBreak*. See "(dt_) Text drawing formatting flags: Chapter 1.

Returns Text height.

Ellipse

Declaration `function Ellipse(DC: HDC; X1, Y1, X2, Y2: Integer): Bool;`

Draws an ellipse centered in the bounding rectangle, whose border is drawn with the current pen and filled with the current brush.

Parameters *DC*: Device context identifier.

X1, Y1: Upper-left corner of the bounding rectangle.

X2, Y2: Lower-left corner of the bounding rectangle.

Returns Non-zero if the ellipse was drawn, zero if not.

EmptyClipboard

Declaration `function EmptyClipboard: Bool;`

Empties the clipboard and frees handles to data in the clipboard. Ownership is assigned to the window that has the clipboard open.

Returns Non-zero if the clipboard is emptied; zero if error.

EnableHardwareInput

Declaration `function EnableHardwareInput(EnableInput: Bool): Bool;`

Disables mouse and keyboard input saving or discarding input as specified by *EnableInput*.

Parameters *EnableInput*: Non-zero to save input; zero to discard input.

Returns Non-zero (default) if input was previously enabled; zero if not.

EnableMenuItem

Declaration `function EnableMenuItem(Menu: HMenu; IDEnableItem, Enable: Word): Bool;`

Enables, disables, or grays a menu item as specified by *Enable*.

Parameters *Menu*: Menu identifier

IDEnableItem: ID or position of menu item or pop-up to be checked.

Enable: A combination of the constants *mf_Command* or *mf_ByPosition* with *mf_Disabled*, *mf_Enabled*, or *mf_Grayed*. See "(MF_) Menu flags" in Chapter 1.

Returns Previous state of menu item; –1 if menu item does not exist.

EnableWindow

Declaration `function EnableWindow(Wnd: HWnd; Enable: Bool): Bool;`

Enables or disables mouse and keyboard input to a window or control.

Parameters *Wnd*: Window to be enabled or disabled.

Enable: Non-zero to enable; zero to disable.

Returns Non-zero if successful; zero if not.

EndDeferWindowPos

Declaration `procedure EndDeferWindowPos(WinPosInfo: THandle);`

Simultaneously updates, in a single screen-refresh cycle, size and position of one or more windows.

Parameters *WinPosInfo*: Multiple window data structure containing update info for multiple windows.

See also *BeginDeferWindowPos, DeferWindowPos*

EndDialog

Declaration `procedure EndDialog(Dlg: HWnd; Result: Integer);`

Terminates a modal dialog box. The value specified by *Result* is returned to the creating *DialogBox* function.

Parameters *Dlg*: Dialog to be destroyed.

Result: Return value.

EndPaint

Declaration `procedure EndPaint(Wnd: HWnd; var Paint: TPaintStruct);`

Signifies end of painting in *Wnd*.

Parameters *Wnd*: Repainted window.

Paint: *TPaintStruct* retrieved from *BeginPaint* function.

EnumChildWindows

Declaration `function EnumChildWindows(WndParent: HWnd; EnumFunc: TFarProc; lParam: Longint): Bool;`

Enumerates child windows of given parent passing the child handle and *lParam* to the callback. Enumeration ends if the callback returns zero or the last child is enumerated.

Parameters *WndParent*: Parent window of child windows to enumerate.

EnumFunc: Callback function's procedure-instance address.

lParam: Value passed to callback function.

Returns Non-zero if all child windows have been enumerated; zero if not.

EnumClipboardFormats

Declaration `function EnumClipboardFormats(Format: Word): Word;`

Enumerates list of available clipboard formats.

Parameters *Format*: A known format or zero for first format in list. The formats are specified by *cf_* constants. See "(cf_) Clipboard formats" in Chapter 1.

Returns Next known clipboard format; zero if end of format list or clipboard not open.

See also *OpenClipboard*.

EnumFonts

Declaration `function EnumFonts(DC: HDC; FaceName: PChar; FontFunc: TFarProc; Data: Pointer): Integer;`

Enumerates available fonts having the specified typeface on a given device. The callback is passed *TLogFont*, *TTextMetric*, *FontType*, and *Data*. Enumeration ends if the callback returns zero or all fonts are enumerated.

Parameters *DC*: Device context identifier.

FaceName: Typeface name (null-terminated) or **nil** to randomly select a one font for each available typeface.

FontFunc: Callback function's procedure-instance address.

Data: Data passed to the callback function.

Returns Last value returned by the callback.

EnumMetaFile

Declaration `function EnumMetaFile(DC: HDC; MF: THandle; CallbackFunc: TFarProc; ClientData: LPByte): Bool;`

Enumerates GDI calls in a metafile passing *DC*, a pointer to the metafile's object handles table, a pointer to a record in the metafile, the number of objects with associated handles in the table, and *ClientData* to the callback. Enumeration ends if the callback returns zero or all GDI calls are enumerated.

Parameters *DC*: Device context associated with metafile.

MF: Metafile identifier.

CallbackFunc: Callback function's procedure instance address.

ClientData: Data passed to callback.

Returns Non-zero if all GDI calls in metafile are enumerated; zero if not.

EnumObjects

Declaration `function EnumObjects(DC: HDC; ObjectType: Integer; ObjectFunc: TFarProc; Data: Pointer): Integer;`

Enumerates object types available on a device passing *TLogPen* or *TLogBrush* and *Data* to the callback. Enumeration ends if the callback returns zero or all objects are enumerated.

Parameters *DC*: Device context identifier.

ObjectType: Can be one of *obj_Brush* or *obj_Pen*. See "obj_ Object type constants" in Chapter 1.

ObjectFunc: Callback function's procedure-instance address.

Data: Data passed to callback.

Returns Last value returned by callback.

EnumProps

Declaration `function EnumProps(Wnd: HWnd; EnumFunc: TFarProc): Integer;`

Enumerates a window's property list passing *Wnd*, nDummy, PSTR, and hData to the callback. Enumeration ends if the callback returns zero or all properties are enumerated.

Parameters *Wnd*: Window identifier.

EnumFunc: Callback function's procedure-instance address.

Returns Last value returned by callback; –1 if no properties.

EnumTaskWindows

Declaration `function EnumTaskWindows(Task: THandle; EnumFunc: TFarProc; lParam: Longint): Bool;`

Enumerates all windows in a task passing the window handle and *lParam* to the callback. Enumeration ends if the callback returns zero or all windows are enumerated.

Parameters *Task*: Task identifier.

EnumFunc: Callback function's procedure-instance address.

lParam: Value passed to callback.

Returns Non-zero if all windows are enumerated; zero if not.

See also *GetCurrentTask*.

EnumWindows

Declaration

```
function EnumWindows(EnumFunc: TFarProc; lParam: Longint): Bool;
```

Enumerates all parent windows on the screen passing the window handle and *lParam* to the callback. Enumeration ends if the callback returns zero or all windows are enumerated.

Parameters *EnumFunc*: Callback function's procedure-instance address.

lParam: Value passed to callback.

Returns Non-zero if all windows are enumerated; zero if not.

EqualRect

Declaration

```
function EqualRect(var Rect1, Rect2: TRect): Bool;
```

Compares the upper-left and lower-right corners of two rectangles for equality.

Parameters *Rect1*, *Rect2*: Rectangles to be compared.

Returns Non-zero if equal, zero if not.

EqualRgn

Declaration

```
function EqualRgn(SrcRgn1, SrcRgn2: HRgn): Bool;
```

Compares two regions for equality.

Parameters *SrcRgn1*, *SrcRgn2*: Regions to be compared.

Returns Non-zero if equal; zero if not.

Escape

Declaration `function Escape(DC: HDC; Escape, Count: Integer; InData, OutData: Pointer): Integer;`

Allows access to device specific facilities not supported by GDI.

Parameters *DC*: Device context identifier.

Escape: *Escape* function.

Count: Bytes of data in *InData*.

InData: The input data structure.

OutData: Structure to receive *Escape* output data or **nil** for no output data.

Returns A positive number if successful, zero if escape is not implemented and negative if error. If error, may be one of the following codes: *sp_Error, sp_OutOfDisk, sp_OutOfMemory, sp_UserAbort*. See "(sp_) Spooler error codes" in Chapter 1.

EscapeCommFunction

Declaration `function EscapeCommFunction(Cid, Func: Integer): Integer;`

Performs extended function specified by *Func* on communications device.

Parameters *Cid*: Communications device.

Func: One of the following constants: *ClrDTR, ClrRTS, ResetDev, SetDTR, SetRTS, SetXoff, SetXon*. See "Escape comm constants" in Chapter 1.

Returns Zero if successful; negative if invalid function code specified.

See also *OpenComm*.

ExcludeClipRect

Declaration `function ExcludeClipRect(DC: HDC; X1, Y1, X2, Y2: Integer): Integer;`

Creates a new clipping region consisting of the existing region minus the specified rectangle.

Parameters *DC*: Device context identifier.

X1, Y1: Upper-left corner of the rectangle.

X2, Y2: Lower-right corner of the rectangle.

Returns New region type: *ComplexRegion, Error, NullRegion, SimpleRegion*. See "Region flags" in Chapter 1.

ExcludeUpdateRgn

Declaration `function ExcludeUpdateRgn(DC: HDC; Wnd: HWnd): Integer;`

Excludes a window's updated region from a clipping region which prevents drawing within invalid areas of the window.

Parameters *DC*: Device context identifier.

Wnd: Window being updated.

Returns New region type: *ComplexRegion, Error, NullRegion, SimpleRegion*. See "Region flags" in Chapter 1.

ExitWindows

Declaration `function ExitWindows(Reserved: DWord; ReturnCode: Word): Bool;`

Initiates standard Windows shutdown procedure. All applications must agree to terminate for Windows to exit. Calls DOS interrupt 21H function 4CH.

Parameters *Reserved*: Set to zero.

ReturnCode: Value passed to DOS (AL).

Returns Zero if one or more applications refuse to terminate.

See also *wm_QueryEndSession, wm_EndSession*

ExtFloodFill

Declaration `function ExtFloodFill(DC: HDC; X, Y: Integer; Color: TColorRef; FillType: Word): Bool;`

Fills an area of a raster display surface in the manner specified by *FillType*. The current brush is used.

Parameters *DC*: Device context identifier.

X, Y: Fill begin point.

Color: *TColorRef* of boundary or area to be filled.

FillType: One of the constants: *FloodFillBorder, FloodFillSurface*. See "Flood fill style flags" in Chapter 1.

Returns Non-zero if successful; zero if not.

See also *FloodFill*.

ExtTextOut

Declaration

```
function ExtTextOut(DC: HDC; X, Y: Integer; Options: Word; Rect: LPRect;
Str: PChar; Count: Word; Dx: LPInteger): Bool;
```

Writes a string using currently selected font within *Rect*.

Parameters *DC*: Device context identifier.

X, Y: Origin of first character cell.

Options: Can be a combination of: *eto_Clipped* and *eto_Opaque*. See "(eto_) ExtTextOut options" in Chapter 1.

ARect: *TRect* or **nil**.

Str: String to write.

Count: Number of characters in string.

Dx: Array of values specifying distance between adjacent cells or 0 for default spacing.

Returns Non-zero if string is drawn; zero if not.

See also *SetTextAlign*.

FatalExit

Declaration

```
procedure FatalExit(Code: Integer);
```

Displays *Code* and stack trace on the computer's auxiliary port. The user is prompted for further action. Used for debugging purposes only.

Parameters *Code*: Displayed error code.

FillRect

Declaration `function FillRect(DC: HDC; var Rect: TRect; Brush: HBrush): Integer;`

Fills a rectangle using *Brush* up to the right and bottom borders.

Parameters *DC*: Device context identifier.

Rect: *TRect* to be filled.

Brush: Fill brush.

Returns Not used.

See also *CreateHatchBrush, CreatePatternBrush, CreateSolidBrush, GetStockObject.*

FillRgn

Declaration `function FillRgn(DC: HDC; Rgn: HRgn; Brush: HBrush): Bool;`

Fills a region using *Brush.*

Parameters *DC*: Device context identifier.

Rgn: Region to be filled.

Brush: Fill brush.

Returns Non-zero if successful; zero if not.

FindAtom

Declaration `function FindAtom(Str: PChar): TAtom;`

Searches the atom table for the atom associated with *Str.*

Parameters *Str*: Search string (null-terminated).

Returns Atom associated with *Str*; 0 if not found in table.

FindResource

Declaration `function FindResource(Instance: THandle; Name, ResType: PChar): THandle;`

Locates a resource in a resource file.

Parameters *Instance*: Module instance whose executable file contains the resource.

Name: Resource name either a null-terminated string or an integer ID.

ResType: One of the following constants, indicating a resource type: *rt_Accelerator, rt_Bitmap, rt_Cursor, rt_Dialog, rt_Font, rt_FontDir, rt_Icon, rt_Menu, rt_RCData, rt_String,* null-terminated string, or an integer ID. See "(RT_) Resource types" in Chapter 1.

Returns Resource identifier; 0 if resource not found.

FindWindow

Declaration `function FindWindow(ClassName, WindowName: PChar): HWnd;`

Finds a top-level parent window with matching *ClassName* and *WindowName*. Does not search child windows.

Parameters *ClassName*: Window's class name (null-terminated or **nil** for all).

WindowName: Window's text caption or 0 for all.

Returns Window handle; 0 if no such window.

FlashWindow

Declaration `function FlashWindow(Wnd: HWnd; Invert: Bool): Bool;`

Flashes a window or icon. An open window's active status is inverted.

Parameters *Wnd*: Window to be flashed.

Invert: Non-zero to flash or zero to return to original state (ignored for icons).

Returns Non-zero if window was active previous to call; zero otherwise.

FloodFill

Declaration `function FloodFill(DC: HDC; X, Y: Integer; Color: TColorRef): Bool;`

Fills display area with current brush bounded by *Color*.

Parameters *DC*: Device context identifier.

X, Y: Position to begin filling.

Color: Boundary color, a *TColorRef*.

Returns Non-zero if successful; zero if not.

FlushComm

Declaration `function FlushComm(Cid, Queue: Integer): Integer;`

Flushes a communication device's transmit or receive queue.

Parameters *Cid*: Communications device to be flushed.

Queue: zero to flush transmit queue or 1 to flush receive queue.

Returns Non-zero if successful; zero if not.

See also *OpenComm*.

FrameRect

Declaration `procedure FrameRect(DC: HDC; var Rect: TRect; Brush: HBrush;`

Draws a one logical unit wide border around a rectangle.

Parameters *DC*: Device context identifier.

Rect: *TRect* defining corners of the rectangle.

Brush: Framing brush.

See also *CreateHatchBrush, CreatePatternBrush, CreateSolidBrush.*

FrameRgn

Declaration `function FrameRgn(DC: HDC; Rgn: HRgn; Brush: HBrush; Width, Height: Integer): Bool;`

Draws a border around a region.

Parameters *DC*: Device context identifier.

Rgn: Region to be closed.

Brush: Framing brush.

Width: Border width in vertical brush strokes (logical units).

Height: Border height in horizontal brush strokes (logical units).

Returns Non-zero if successful; zero if not.

FreeLibrary

Declaration `procedure FreeLibrary(LibModule: THandle);`

Invalidates *LibModule* and frees associated memory if the module is no longer referenced.

Parameters *LibModule*: Loaded library module.

FreeModule

Declaration `function FreeModule(Module: THandle): Bool;`

Invalidates *Module* and frees associated memory if the module is no longer referenced.

Parameters *Module*: Loaded module identifier.

Returns Not used.

FreeProcInstance

Declaration `procedure FreeProcInstance(Proc: TFarProc);`

Frees a function's procedure-instance address.

Parameters *Proc*: Function's procedure-instance address to be freed.

See also *MakeProcInstance*.

FreeResource

Declaration `function FreeResource(ResData: THandle): Bool;`

Invalidates *ResData* and frees associated memory if the resource is no longer referenced.

Parameters *ResData*: Resource data identifier.

Returns Zero if successful; non-zero if not.

See also *LoadResource*.

GetActiveWindow

Declaration `function GetActiveWindow: HWnd;`

Retrieves the handle of the window that has the current input focus.

Returns Active window identifier.

See also *SetActiveWindow*.

GetAspectRatioFilter

Declaration `function GetAspectRatioFilter(DC: HDC): Longint;`

Retrieves the aspect ratio used by the current aspect ratio filter.

Parameters *DC*: Device context containing the specified aspect ratio.

Returns: Aspect ratio where the x- and y-coordinates are contained in the high and low word, respectively.

See also *SetMapperFlags*.

GetAsyncKeyState

Declaration `function GetAsyncKeyState(Key: Integer): Integer;`

Determines the state of a virtual key.

Parameters *Key*: Virtual-key code.

Returns If MSB set *Key* is down and if LSB set *Key* was pressed after a preceding call to the function.

GetAtomHandle

Declaration `function GetAtomHandle(AnAtom: TAtom): THandle;`

Retrieves the string that corresponds to the specified atom.

Parameters *AnAtom*: Atom identifier.

Returns Local memory handle to atom string; zero atom does not exist.

GetAtomName

Declaration `function GetAtomName(AnAtom: TAtom; Buffer: PChar; Size: Integer): Word;`

Copies associated atom string into *Buffer*.

Parameters *AnAtom*: Atom identifier.

Buffer: Buffer to receive atom string.

Size: Buffer size (in bytes).

Returns Number of bytes copied into *Buffer*; zero if invalid atom specified.

GetBitmapBits

Declaration `function GetBitmapBits(Bitmap: HBitmap; Count: Longint; Bits: Pointer): Longint;`

Copies the bits of a bitmap into *Bits*.

Parameters *Bitmap*: Bitmap identifier.

Count: Number of bytes to copy.

Bits: Byte array which conforms to a structure where horizontal scan lines are multiples of 16 bits.

Returns Actual number of bytes in the bitmap; zero if error.

GetBitmapDimension

Declaration `function GetBitmapDimension(Bitmap: HBitmap): Longint;`

Retrieves the height and width of a bitmap.

Parameters *Bitmap*: Bitmap identifier.

Returns Height and width (in tenths of millimeters) in high and low word, respectively.

See Also *SetBitmapDimension*

GetBkColor

Declaration `function GetBkColor(DC: HDC): Longint;`

Retrieves the current device background color.

Parameters *DC*: Device context identifier.

Returns RGB color value.

GetBkMode

Declaration `function GetBkMode(DC: HDC): Integer;`

Retrieves the current device background mode, used for text, hatched brushes, and non-solid line pen styles.

Parameters *DC*: Device context identifier.

Returns One of the constants *Opaque* or *Transparent*. See "Background modes" in Chapter 1.

GetBrushOrg

Declaration `function GetBrushOrg(DC: HDC): Longint;`

Retrieves the current device brush origin.

Parameters *DC*: Device context identifier.

Returns X- and y-coordinates in low and high word, respectively.

GetBValue

Declaration `function GetBValue(RGBColor: Longint): Byte;`

Extracts the blue intensity value from an RGB color value.

Parameters *RGBColor*: An RGB color value.

Returns The blue intensity value, from 0 to 255.

GetCapture

Declaration `function GetCapture: HWnd;`

Retrieves the window that is currently receiving all mouse input.

Returns Window that has the mouse capture; 0 if no window.

See also *SetCapture*.

GetCaretBlinkTime

Declaration `function GetCaretBlinkTime: Word;`

Retrieves caret blink (time between flashes of the caret).

Returns Blink rate (in milliseconds).

GetCaretPos

Declaration `procedure GetCaretPos(var Point: TPoint);`

Retrieves the current caret position (in client coordinates).

Parameters *Point*: Receiving *TPoint* structure.

GetCharWidth

Declaration `function GetCharWidth(DC: HDC; FirstChar, LastChar: Word; var Buffer): Bool;`

Retrieves individual character widths for the specified group of consecutive characters.

Parameters *DC*: Device context identifier.

FirstChar: First character in consecutive character group.

LastChar: Last character in consecutive character group.

Buffer: Receiving integer array for width values.

Returns Non-zero if successful; zero if not.

GetClassInfo

Declaration `function GetClassInfo(Instance: THandle; ClassInfo: PChar; var WndClass: TWndClass): Bool;`

Retrieves class information for the specified class. *TWndClass* fields *lpszClassName, lpszMenuName,* and *hInstance* are not returned.

Parameters *Instance*: Application instance that created the class or 0 for a predefined Windows class.

ClassName: Class name (null-terminated) or ID.

WndClass: *TWndClass* to receive class information.

Returns Non-zero if successful; zero if matching class not found.

GetClassLong

Declaration `function GetClassLong(Wnd: HWnd; Index: Integer): Longint;`

Retrieves the long value at *Index* from a window's *TWndClass* structure. Positive byte offsets (from zero) to access extra class bytes.

Parameters *Wnd*: Window identifier.

Index: Byte offset or the constant *gcl_WndProc*. See "(gcl_) Class field offsets" in Chapter 1.

Returns Value retrieved.

GetClassName

Declaration `function GetClassName(Wnd: HWnd; ClassName: PChar; MaxCount: Integer): Integer;`

Retrieves a window's class name.

Parameters *Wnd*: Window identifier.

ClassName: Buffer to receive class name.

MaxCount: Size of buffer.

Returns Actual number of characters copied; zero if error.

GetClassWord

Declaration `function GetClassWord(Wnd: HWnd, Index: Integer): Word;`

Retrieves the word value at *Index* from a window's *TWndClass* structure. Positive byte offsets (from zero) to access extra class bytes.

Parameters *Wnd*: Window identifier.

Index: Byte offset or *gcw_CBClsExtra, gcw_CBWndExtra, gcw_HBrBackground, cgw_HCursor, gcw_HIcon, gcw_HModule, gcw_Style.* See "(gcw_) Class field offsets" in Chapter 1.

Returns Value retrieved.

GetClientRect

Declaration `procedure GetClientRect(Wnd: HWnd; var Rect: TRect);`

Retrieves a window's client coordinates.

Parameters *Wnd*: Window identifier.

Rect: *TRect* to receive client coordinates.

GetClipboardData

Declaration `function GetClipboardData(Format: Word): THandle;`

Retrieves clipboard data in specified format. The returned memory block is controlled by the clipboard.

Parameters *Format*: Clipboard data format. One of the *cf_* constants. See "(cf_) Clipboard formats" in Chapter 1.

Returns Memory block containing clipboard data; 0 if error.

See also *SetClipboardData.*

GetClipboardFormatName

Declaration `function GetClipboardFormatName(Format: Word; FormatName: PChar; MaxCount: Integer): Integer;`

Retrieves a registered format name from the clipboard.

Parameters *Format*: Clipboard format. One of the CF_ constants. See "(CF_) Clipboard formats" in Chapter 1.

FormatName: Receiving buffer.

MaxCount: Size of buffer.

Returns Actual length of copied string; zero if invalid format requested.

GetClipboardOwner

Declaration `function GetClipboardOwner: HWnd;`

Retrieves the window that currently owns the clipboard.

Returns Owning window; zero if no owner.

GetClipboardViewer

Declaration `function GetClipboardViewer: HWnd;`

Retrieves the first window in the clipboard-viewer chain.

Returns Window currently responsible for displaying the clipboard; 0 if no viewer.

GetClipBox

Declaration `function GetClipBox(DC: HDC; var Rect: TRect): Integer;`

Retrieves the tightest bounding rectangle around the current clipping boundary.

Parameters *DC*: Device context identifier.

Rect: Receiving *TRect* structure.

Returns The clipping region's type: *ComplexRegion*, *Error*, *NullRegion*, or *SimpleRegion*. See "Region flags" in Chapter 1.

GetCodeHandle

Declaration
```
function GetCodeHandle(Proc: TFarProc): THandle;
```

Retrieves the code segment (loading it if necessary) that contains the specified function.

Parameters *Proc*: Function's procedure-instance address.

Returns Code segment that contains the function.

GetCodeInfo

Declaration
```
procedure GetCodeInfo(Proc: TFarProc; SegInfo: Pointer);
```

Retrieves information about the code segment containing *Proc*.

Parameters *Proc*: Function address or module handle and segment number.

SegInfo: Array of four 32-bit values.

GetCommError

Declaration
```
function GetCommError(Cid: Integer; var Stat: TComStat): Integer;
```

Clears a devices communications error.

Parameters *Cid*: Communications device.

Stat: *TComStat* to receive device status information or **nil**.

Returns One of the following error codes: *ce_Break*, *ce_CTSTo*, *ce_DNS*, *ce_DSRTo*, *ce_Frame*, *ce_IOE*, *ce_Mode*, *ce_OOP*, *ce_Overrun*, *ce_PTO*, *ce_RLSDTO*, *ce_RxOver*, *ce_RxParity*, *ce_TxFull*. See "(CE_) Comm error flags" in Chapter 1.

See also *OpenComm*.

GetCommEventMask

Declaration `function GetCommEventMask(Cid, EvtMask: Integer): Word;`

Retrieves a device's current event mask and then clears it.

Parameters *Cid*: Communications device.

EvtMask: Events to be enabled.

Returns Current event mask value.

See also *OpenComm, SetCommEventMask.*

GetCommState

Declaration `function GetCommState(Cid: Integer; var DCB: TDCB): Integer;`

Retrieves a device's device control block.

Parameters *Cid*: Communications device.

DCB: *TDCB* to receive the current device control block.

Returns Zero if successful; negative if not.

See also *OpenComm.*

GetCurrentPDB

Declaration `function GetCurrentPDB: Word;`

Retrieves the current DOS Program Data Base (PDB), also known as the Program Segment Prefix.

Returns Current PDB paragraph address or selector.

GetCurrentPosition

Declaration `function GetCurrentPosition(DC: HDC): Longint;`

Retrieves the logical coordinates of the current position.

Parameters *DC*: Device context identifier.

Returns x and y coordinate in the low and high word, respectively.

GetCurrentTask

Declaration `function GetCurrentTask :THandle;`

Retrieves handle of the currently executing task.

Returns Task identifier if successful; 0 if not.

GetCurrentTime

Declaration `function GetCurrentTime: Longint;`

Retrieves elapsed time since the system was rebooted.

Returns Current time (in milliseconds).

GetCursorPos

Declaration `procedure GetCursorPos(var Point: TPoint);`

Retrieves the screen coordinates of the cursor's current position.

Parameters *Point*: Receiving *TPoint* structure.

GetDC

Declaration `function GetDC(Wnd: HWnd): HDC;`

Retrieves a display context for performing GDI operations in a window's client area.

Parameters *Wnd*: Window identifier.

Returns Display context identifier; 0 if error.

See also *ReleaseDC*.

GetDCOrg

Declaration `function GetDCOrg(DC: HDC): Longint;`

Retrieves a device's final translation origin (in screen coordinates) which is the offset used by Windows to translate device coordinates to client coordinates.

Parameters *DC*: Device context identifier.

Returns X and y coordinate in the low and high word, respectively.

GetDesktopWindow

Declaration `function GetDesktopWindow: HWnd;`

Retrieves the handle to Windows desktop window.

Returns Windows desktop window identifier.

GetDeviceCaps

Declaration `function GetDeviceCaps(DC: HDC; Index: Integer): Integer;`

Retrieves a display's device-specific information.

Parameters *DC*: Device context identifier.

Index: Item to return. See "Device capabilities" in Chapter 1.

Returns Desired item value.

GetDialogBaseUnits

Declaration `function GetDialogBaseUnits: Longint;`

Retrieves dialog base units. The base width represents the average system font width. The actual dialog unit is 1/4 and 1/8 the returned base width and height unit, respectively.

Returns Height and width (in pixels) base units in high and low word, respectively.

GetDIBits

Declaration
```
function GetDIBits(DC: HDC; Bitmap: THandle; StartScan, NumScans: Word;
Bits: Pointer; var BitInfo: TBitmapInfo; Usage: Word): Integer;
```

Copies a bitmap, in device independent format, to *Bits*.

Parameters *DC*: Device context identifier.

Bitmap: Source bitmap identifier.

StartScan: First scan line.

NumScans: Number of lines to copy.

Bits: Buffer to receive bitmap or **nil** to fill *BitsInfo*.

BitInfo: *TBitmapInfo* containing color format and dimension of bitmap.

Usage: Indicates the source of colors. One of the following constants: *DIB_Pal_Colors, DIB_RGB_Colors*. See "(DIB_) Color table identifiers" in Chapter 1.

Returns Number of scan lines copied; zero if error.

GetDlgCtrlID

Declaration `function GetDlgCtrlID(Wnd: HWnd): Integer;`

Retrieves a control window's ID value.

Parameters *Wnd*: Control identifier.

Returns Control's numeric identifier; zero if error.

GetDlgItem

Declaration `function GetDlgItem(Dlg: HWnd; IDDlgItem: Integer): HWnd;`

Retrieves a control handle contained in the specified dialog box.

Parameters *Dlg*: Dialog box containing the control.

IDDlgItem: Control ID.

Returns Control identifier; 0 if specified control does not exist.

GetDlgItemInt

Declaration `function GetDlgItemInt(Dlg: HWnd; IDDlgItem: Integer; Translate: LPBool; Signed: Bool): Word;`

Translates a control's text, in a dialog box, into an integer value. Preceding spaces are stripped.

Parameters *Dlg*: Dialog box identifier.

IDDlgItem: Item ID.

Translate: Returned *Bool* value; zero if translation error.

Signed: Treat retrieved value as signed.

Returns Translated value.

See also *wm_GetText*.

GetDlgItemText

Declaration `function GetDlgItemText(Dlg: HWnd; IDDlgItem: Integer; Str: PChar; MaxCount: Integer): Integer;`

Retrieves a control's text.

Parameters *Dlg*: Dialog box identifier.

IDDlgItem: Item ID.

Str: Buffer to receive text.

MaxCount: Size of buffer.

Returns Actual number of characters copied.

See also *wm_GetText*.

GetDOSEnvironment

Declaration `function GetDOSEnvironment: PChar;`

Retrieves the current task's DOS environment string.

Returns Task's environment string.

GetDoubleClickTime

Declaration `function GetDoubleClickTime: Word;`

Retrieves the maximum time between a series of two mouse clicks that constitutes a double-click.

Returns Current double-click time (in milliseconds).

GetDriveType

Declaration `function GetDriveType(Drive: Integer): Word;`

Determines whether *Drive* is removeable, fixed, or remote.

Parameters *Drive*: Drive to check (i.e. A: is 0, B: is 1, etc).

Returns *Drive_Removeable, Drive_Fixed, Drive_Remote* or zero if cannot determine; 1 if does not exist. See "(DRIVE_) Drive types" in Chapter 1.

GetEnvironment

Declaration `function GetEnvironment(PortName, Environ: PChar; MaxCount: Word): Integer;`

Retrieves the current environment for the device attached to the system port.

Parameters *PortName*: Port name (null-terminated).

Environ: Buffer to receive environment (first field must contain device name) or **nil** to return required size.

MaxCount: Size of buffer.

Returns Actual number of bytes copied; zero if environment cannot be found.

GetFocus

Declaration `function GetFocus: HWnd;`

Retrieves the window that currently has the input focus.

Returns Window identifier if successful; 0 if not.

GetFreeSpace

Declaration `function GetFreeSpace(Flag: Word): Longint;`

Retrieves the amount of free memory in the global heap.

Parameters *Flag*: The constant *gmem_Not_Banked* to scan below bank line or zero to scan above; ignored for non-EMS systems. See "(GMEM_) Global memory flags" in Chapter 1.

Returns Available memory (in bytes).

See also *GlobalCompact*.

GetGValue

Declaration `function GetGValue(RGBColor: Longint): Byte;`

Extracts the green intensity value from an RGB color value.

Parameters *RGBColor*: An RGB color value.

Returns The green intensity value, from 0 to 255.

GetInputState

Declaration `function GetInputState: Bool;`

Determines whether the system queue currently contains mouse, keyboard, or timer events.

Returns Non-zero if input detected; zero if not.

GetInstanceData

Declaration `function GetInstanceData(Instance: THandle; Data, Count: Word): Integer;`

Copies a previous instance data into *Data*.

Parameters *Instance*: Previous application's instance identifier.

Data: Receiving buffer.

Count: Size of buffer.

Returns Actual number of bytes copied.

GetKBCodePage

Declaration `function GetKBCodePage: Integer;`

Retrieves the currently loaded OEM/ANSI table.

Returns Current code page; (437) USA, (850) International, (860) Portugal, (861) Iceland, (863) French Canadian, (865) Norway/Denmark.

GetKeyboardState

Declaration `procedure GetKeyboardState(var KeyState: Byte);`

Copies the virtual-keyboard key set status into *KeyState*. If high-bit of byte is 1 key is down. If low-bit is 1 key was pressed odd number of times since system startup.

Parameters *KeyState*: 256 byte character array.

GetKeyboardType

Declaration `function GetKeyboardType(TypeFlag: Integer): Integer;`

Retrieves the system keyboard type.

Parameters *TypeFlag*: 0 (keyboard type), 1 (keyboard subtype), 2 (number of function keys—FK) .

Returns 1 (PC/XT, 10 FKs), 2 (Olivetti M24, 12 FKs), 3 (AT, 10 FKs), 4 (Enhanced, 12 FKs), 5 (Nokia 1050, 10 FKs), 6 (Nokia 9140, 24 FKs).

GetKeyNameText

Declaration `function GetKeyNameText(lParam: Longint; Buffer: PChar; Size: Integer): Integer;`

Retrieves key name string for keys longer than a single character.

Parameters *lParam*: Long parameter to *wm_KeyDown* message.

Buffer: Receiving buffer.

Size: Size of buffer.

Returns Actual number of bytes copied.

GetKeyState

Declaration `function GetKeyState(VirtKey: Integer): Integer;`

Retrieves whether the state of a virtual key is up, down, or toggled.

Parameters *VirtKey*: Virtual key.

Returns Key is down if high-order bit is 1 and key is toggled if low-order bit is 1.

GetLastActivePopup

Declaration `function GetLastActivePopup(WndOwner: HWnd): HWnd;`

Retrieves the most-recently active popup.

Parameters *WndOwner*: Owning parent window of popup.

Returns Popup window identifier; *WndOwner*.

GetMapMode

Declaration `function GetMapMode(DC: HDC): Integer;`

Retrieves current mapping mode.

Parameters *DC*: Device context identifier.

Returns Mapping mode, a MM_ constant. See "(MM_) Mapping modes" in Chapter 1.

See also *SetMapMode*.

GetMenu

Declaration `function GetMenu(Wnd: HWnd): HMenu;`

Retrieves a window's menu handle.

Parameters *Wnd*: Owning window of menu.

Returns Menu identifier; 0 if no menu; undefined if *Wnd* is a child window.

GetMenuCheckMarkDimensions

Declaration `function GetMenuCheckMarkDimensions: Longint;`

Retrieves the dimensions of the default checkmark bitmap which is displayed next to checked menu items.

Returns Height and width (in pixels) in the high and low order word, respectively.

See also *SetMenuItemBitmaps*.

GetMenuItemCount

Declaration `function GetMenuItemCount(Menu: HMenu): Word;`

Retrieves the number of top-level menus and menu items in the specified menu.

Parameters *Menu*: The menu identifier.

Returns The number of menu items, if successful; –1 if unsuccessful.

GetMenuItemID

Declaration `function GetMenuItemID(Menu: HMenu; Pos: Integer): Word;`

Retrieves the menu item numeric ID located at the specified position in the menu.

Parameters *Menu*: Popup menu identifier.

Pos: Item position, starting at zero, in menu.

Returns Item ID if successful; 0 if item is a popup; –1 if error.

GetMenuState

Declaration `function GetMenuState(Menu: HMenu; ID, Flags: Word): Word;`

Retrieves state information for the specified menu item.

Parameters *Menu*: Menu or popup menu identifier.

ID: Menu item ID.

Flags: One of the menu constants: *mf_ByPosition, mf_ByCommand*. See "(MF_) Menu flags" in Chapter 1.

Returns Flags mask of the following values: *mf_Checked, mf_Disabled, mf_Enabled, mf_MenuBarBreak, mf_MenuBreak, mf_Separator, mf_Unchecked*; if popup, high-order byte contains number of items; –1 if ID is invalid. See "(MF_) Menu flags" in Chapter 1.

GetMenuString

Declaration `function GetMenuString(Menu: HMenu; IDItem: Word; Str: PChar; MaxCount: Integer; Flag: Word): Integer;`

Copies a menu item's label into *Str*. The copied label is null-terminated.

Parameters *Menu*: Menu identifier.

IDItem: Menu item ID.

Str: Receiving buffer.

MaxCount: Size of buffer.

Flag: One of the menu constants: *mf_ByPosition, mf_ByCommand*. See "(MF_) Menu flags" in Chapter 1.

Returns Actual number of bytes copied.

GetMessage

Declaration

```
function GetMessage(var Msg: TMsg; Wnd: HWnd; MsgFilterMin, MsgFilterMax:
Word): Bool;
```

Retrieves a message, within filter range, from the applications message queue. Yields control to other applications if no messages available or *wm_Paint*, or *wm_Timer* is the next message.

Parameters *Msg*: Receiving *TMsg* structure.

Wnd: Message destination window or 0 for all windows in application.

MsgFilterMin: Zero for no filtering or *wm_KeyFirst* for keyboard only or *wm_MouseFirst* for mouse only.

MsgFilterMax: Zero for no filtering or *wm_KeyLast* for keyboard only or *wm_MouseLast* for mouse only.

Returns Non-zero if not *wm_Quit* message; zero otherwise.

GetMessagePos

Declaration

```
function GetMessagePos: Longint;
```

Retrieves the cursor position for the last message obtained from *GetMessage*.

Returns X- and y-coordinates in low and high words, respectively.

GetMessageTime

Declaration `function GetMessageTime: Longint;`

Retrieves the elapsed time since system reboot for the last message obtained from *GetMessage*.

Returns Message time (in milliseconds).

GetMetaFile

Declaration `function GetMetaFile(FileName: PChar): THandle;`

Creates a handle for the named metafile.

Parameters *FileName*: Metafile DOS filename (null-terminated).

Returns Metafile identifier if successful; 0 if not.

GetMetaFileBits

Declaration `function GetMetaFileBits(MF: THandle): THandle;`

Obtains a read-only global memory block containing the metafile as a collection of bits. Used to determine size and save as metafile.

Parameters *MF*: Memory metafile identifier, becomes invalid after call.

Returns Global memory block if successful; 0 if not.

GetModuleFileName

Declaration `function GetModuleFileName(Module: THandle; FileName: PChar; Size: Integer): Integer;`

Retrieves full pathname (null-terminated) of the executable file for the specified module.

Parameters *Module*: Module identifier.

FileName: Receiving buffer.

Size: Size of buffer.

Returns Actual number of bytes copied.

GetModuleHandle

Declaration `function GetModuleHandle(ModuleName: PChar): THandle;`

Retrieves a module's handle.

Parameters *ModuleName*: Module name (null-terminated).

Returns Module identifier if successful; 0 if not.

GetModuleUsage

Declaration `function GetModuleUsage(Module: THandle): Integer;`

Retrieves a module's reference count.

Parameters *Module*: Module identifier.

Returns Reference count value.

GetNearestColor

Declaration `function GetNearestColor(DC: HDC; Color: TColorRef): Longint;`

Obtains the closest matching logical color to *Color* that the device can support.

Parameters *DC*: Device context identifier.

Color: *TColorRef* to be matched.

Returns RGB solid color.

GetNearestPaletteIndex

Declaration `function GetNearestPaletteIndex(Palette: HPalette; Color: TColorRef): Word;`

Obtains the closest matching color in a logical-palette to *Color*.

Parameters *Palette*: Logical palette identifier.

Color: *TColorRef* to be matched.

Returns Logical-palette entry index.

GetNextDlgGroupItem

Declaration `function GetNextDlgGroupItem(Dlg: HWnd; Ctrl: HWnd; Previous: Bool): HWnd;`

Retrieves the next or previous *ws_Group* style control from *Ctrl*. The search is cyclical.

Parameters *Dlg*: Dialog box identifier.

Ctrl: Search start control identifier.

Previous: Zero to search for previous control or non-zero to search for next.

Returns Control identifier.

GetNextDlgTabItem

Declaration `function GetNextDlgTabItem(Dlg: HWnd; Ctrl: HWnd; Previous: Bool): HWnd;`

Retrieves the next or previous *ws_TabStop* style control from *Ctrl*. The search is cyclical.

Parameters *Dlg*: Dialog box identifier.

Ctrl: Search start control identifier.

Previous: Zero to search for previous control or non-zero to search for next.

Returns Control identifier.

GetNextWindow

Declaration `function GetNextWindow(Wnd: HWnd; Flag: Word): HWnd;`

Retrieves next or previous window from *Wnd*. If top-level window searches for next top-level window or if child window searches for next child window.

Parameters *Wnd*: Window identifier.

Flag: One of the following constants: *gw_HWndNext* or *gw_HWndPrev*. See "(GW_) Get window constants" in Chapter 1.

Returns Window identifier.

GetNumTasks

Declaration `function GetNumTasks: Word;`

Retrieves number of tasks that are currently executing in the system.

Returns Number of currently executing tasks.

GetObject

Declaration `function GetObject(hObject: THandle; Count: Integer; ObjectPtr: Pointer): Integer;`

Fills a buffer with data that defines a logical object. Only returns number of entries for logical palettes.

Parameters *hObject*: Object identifier.

Count: Size of buffer.

ObjectPtr: Receiving buffer; *TLogPen, TLogBrush, TLogFont, TBitmap,* or an integer.

Returns Actual number of bytes copied; zero if error.

See also *GetBitmapBits, GetPaletteEntries.*

GetPaletteEntries

Declaration `function GetPaletteEntries(Palette: HPalette; StartIndex, NumEntries: Word; var PaletteEntries): Word;`

Retrieves the specified range of palette entries and copies them to *PaletteEntries*.

Parameters *Palette*: Logical palette identifier.

StartIndex: First entry.

NumEntries: Number of entries.

PaletteEntries: *TPaletteEntry* array to receive the palette entries.

Returns Actual number of entries retrieved; zero if error.

GetParent

Declaration `function GetParent(Wnd: HWnd): HWnd;`

Retrieves a window's parent window handle.

Parameters *Wnd*: Window identifier.

Returns Parent window identifier; 0 if no parent window.

GetPixel

Declaration `function GetPixel(DC: HDC; X, Y; Integer): Longint;`

Retrieves the RGB color at the specified point.

Parameters *DC*: Device context identifier.

X, Y: Point to be examined.

Returns RGB color value; –1 if point not in clipping region.

GetPolyFillMode

Declaration `function GetPolyFillMode(DC: HDC): Integer;`

Retrieves the current polygon filling mode.

Parameters *DC*: Device context identifier.

Returns Polygon filling mode. One of the constants *Alternate* or *Winding*. See "PolyFill modes" in Chapter 1.

GetPriorityClipboardFormat

Declaration `function GetPriorityClipboardFormat(var PriorityList; Count: Integer): Integer;`

Retrieves the first clipboard format in *PriorityList* for which data exists.

Parameters *PriorityList*: Integer array containing clipboard formats in priority order. The formats are CF_ constants. See "(CF_) Clipboard formats" in Chapter 1.

Count: Size of *PriorityList*.

Returns Highest priority format in list; –1 if no match.

GetPrivateProfileInt

Declaration `function GetPrivateProfileInt(ApplicationName, KeyName: PChar; Default: Integer; FileName: PChar): Word;`

Retrieves an integer key value from the specified initialization file.

Parameters *ApplicationName*: Application heading name in *FileName*.

KeyName: Key name in *FileName*.

Default: Default value if *KeyName* not found.

FileName: Initialization filename in Windows directory.

Returns Key value; zero if negative or not an integer;.

GetPrivateProfileString

Declaration `function GetPrivateProfileString(ApplicationName, KeyName, Default, ReturnedString: PChar; Size: Integer; FileName: PChar): Integer;`

Retrieves a string key value from the specified initialization file.

Parameters *ApplicationName*: Application heading name in *FileName*.

KeyName: Key name in *FileName* or **nil** to obtain list of key names.

Default: Default value if *KeyName* not found.

ReturnedString: Receiving buffer.

Size: Size of buffer.

FileName: Initialization filename in Windows directory.

Returns Actual number of characters copied.

GetProcAddress

Declaration `function GetProcAddress(Module: THandle; ProcName: PChar): TFarProc;`

Retrieves the address of an exported library function.

Parameters *Module*: Library module.

ProcName: Function name (null-terminated) or ordinal name.

Returns Function's entry point if successful; zero if not.

GetProfileInt

Declaration `function GetProfileInt(AppName, KeyName: PChar; Default: Integer): Integer;`

Retrieves an integer key value from Windows WIN.INI file.

Parameters *AppName*: Application heading name.

KeyName: Search key name.

Default: Default value if *KeyName* not found.

Returns Key value; zero if negative or not an integer.

GetProfileString

Declaration

```
function GetProfileString(AppName, KeyName, Default, ReturnedString:
PChar; Size: Integer): Integer;
```

Retrieves a string key value from Windows WIN.INI file.

Parameters *AppName*: Application heading name.

KeyName: Search key name or **nil** to obtain all key names associated with *AppName*.

Default: Default value if *KeyName* not found.

ReturnedString: Receiving buffer.

Size: Size of buffer.

Returns Actual number of characters copied.

GetProp

Declaration

```
function GetProp(Wnd: HWnd; Str: PChar): THandle;
```

Retrieves the associated data handle from a windows property list.

Parameters *Wnd*: Window identifier.

Str: String (null-terminated) or atom.

Returns Data handle if property list contains *Str*; 0 if not.

See also *AddAtom*.

GetRgnBox

Declaration

```
function GetRgnBox(Rgn: HRgn; var Rect: TRect): Integer;
```

Retrieves a region's bounding rectangle.

Parameters *Rgn*: Region identifier.

Rect: Receiving *TRect*.

Returns Regions type, one of *ComplexRegion, NullRegion, SimpleRegion*; zero if invalid region. See "Region flags" Chapter 1, "Windows styles and constants."

GetROP2

Declaration
```
function GetROP2(DC: HDC): Integer;
```

Retrieves the current drawing mode.

Parameters *DC*: Raster device context.

Returns Drawing mode. One of the R2_ constants. See "(R2_) Binary raster operations" Chapter 1.

See also *SetROP2*.

GetRValue

Declaration
```
function GetRValue(RGBColor: Longint): Byte;
```

Extracts the red intensity value from an RGB color value.

Parameters *RGBColor*: An RGB color value.

Returns The red intensity value, from 0 to 255.

GetScrollPos

Declaration
```
function GetScrollPos(Wnd: HWnd; Bar: Integer): Integer;
```

Retrieves current position of a scroll bar thumb relative to the current scrolling range.

Parameters *Wnd*: Window containing scroll bar.

Bar: One of the following constants: *sb_Ctl, sb_Horz, sb_Vert*. See "(SB_) Scroll bar constants" in Chapter 1.

Returns Current scroll bar thumb position.

GetScrollRange

Declaration `procedure GetScrollRange(Wnd: HWnd; Bar: Integer; var MinPos, MaxPos: Integer);`

Retrieves a scroll bar's minimum and maximum positions.

Parameters *Wnd*: Window containing scroll bar.

Bar: One of the following constants: *sb_Ctl, sb_Horz, sb_Vert*. See "(SB_) Scroll bar constants" in Chapter 1.

MinPos: Integer to receive minimum position.

MaxPos: Integer to receive maximum position.

GetStockObject

Declaration `function GetStockObject(Index: Integer): THandle;`

Retrieves a handle to a predefined stock pen, brush, or font.

Parameters *Index*: One of the following constants: *Black_Brush, DkGray_Brush, Gray_Brush, Hollow_Brush, LtGray_Brush, Null_Brush, White_Brush, Black_Pen, Null_Pen, White_Pen, ANSI_Fixed_Font, ANSI_Var_Font, Device_Default_Font, OEM_Fixed_Font, System_Font, System_Fixed_Font, Default_Palette*. See "Stock logical objects" in Chapter 1.

Returns Desired logical object identifier if successful; 0 if not.

GetStretchBltMode

Declaration `function GetStretchBltMode(DC: HDC): Integer;`

Retrieves the current stretching mode.

Parameters *DC*: Device context identifier.

Returns One of the constants: *WhiteOnBlack, BlackOnWhite*, or *ColorOnColor*. See "StretchBlt modes" in Chapter 1.

See also *SetStretchBltMode*.

GetSubMenu

Declaration `function GetSubMenu(Menu: HMenu; Pos: Integer): HMenu;`

Retrieves a popup menu's handle.

Parameters *Menu*: Menu identifier.

Pos: Position of popup menu in *Menu*.

Returns Popup menu identifier; 0 if no popup exists at *Pos*.

GetSysColor

Declaration `function GetSysColor(Index: Integer): Longint;`

Retrieves a Windows display element's current color.

Parameters *Index*: Display element.

Returns RGB color value.

See also *SetSysColor*.

GetSysModalWindow

Declaration `function GetSysModalWindow: HWnd;`

Retrieves the current system-modal window's handle.

Returns System-modal window identifier if one is present; 0 if not.

GetSystemDirectory

Declaration `procedure GetSystemDirectory(Buffer: PChar; Size: Word);`

Obtains the Windows system subdirectory pathname.

Parameters *Buffer*: Receiving buffer.

Size: Size of *Buffer* (at least 144 characters).

GetSystemMenu

Declaration `function GetSystemMenu(Wnd: HWnd; Revert: Bool): HMenu;`

Retrieves a window's system menu for copying and modification.

Parameters *Wnd*: Window identifier.

Revert: Zero to return handle to a copy of the system menu or non-zero to return handle to original system menu.

Returns System menu identifier; 0 if *Revert* is non-zero and system menu has not been modified.

See also *AppendMenu, InsertMenu, ModifyMenu.*

GetSystemMetrics

Declaration `function GetSystemMetrics(Index: Integer): Integer;`

Retrieves system metrics, such as pixel width and heights of various display elements, mouse status, and Windows debug version.

Parameters *Index*: One of the SM_ constants. See "(SM_) System metrics codes" in Chapter 1.

Returns Requested system metric value.

GetSystemPaletteEntries

Declaration
```
function GetSystemPaletteEntries(DC: HDC; StartIndex, NumEntries: Word;
var PaletteEntries: TPaletteEntry): Word;
```

Retrieves a range of palette entries from the system palette.

Parameters *DC*: Device context identifier.

StartIndex: First entry to retrieve.

NumEntries: Number of entries to retrieve.

PaletteEntries: Receiving *TPaletteEntry* array.

Returns Actual number of entries retrieved; zero if error.

GetSystemPaletteUse

Declaration `function GetSystemPaletteUse(DC: HDC): Word;`

Determines whether an application has full access to the system palette.

Parameters *DC*: Device context identifier.

Returns One of the constants *syspal_NoStatic* or *syspal_Static*. See "(syspal_) System palette flags" in Chapter 1.

See also *SetSystemPaletteUse*.

GetTabbedTextExtent

Declaration `function GetTabbedTextExtent(DC: HDC; Str: PChar; Count, TabPositions: Integer; var TabStopPositions): Longint;`

Computes the height and width (in pixels) of *Str* using the currently selected font. Tabs are expanded as specified.

Parameters *DC*: Device context identifier.

Str: Line of text.

Count: Number of characters in *Str*.

TabPositions: Number of tab-stop positions in *TabStopPositions* or zero and tabs are expanded eight average character widths.

TabStopPositions: Integer array containing increasing order of tab-stop positions (in pixels).

Returns Height and width in high and low order word, respectively.

GetTempDrive

Declaration `function GetTempDrive(DriveLetter: Char): Char;`

Retrieves the disk-drive that will provide optimal access time for temporary file operations.

Parameters *DriveLetter*: Disk-drive letter or zero to return current drive.

Returns Disk-drive letter.

GetTempFileName

Declaration
```
function GetTempFileName(DriveLetter: Char; PrefixString: PChar; Unique:
Word; TempFileName: PChar): Integer;
```

Derives a unique temporary filename whose pathname is either the root directory or that specified by the TEMP environment variable.

Parameters *DriveLetter*: Suggested drive or *tf_ForceDrive* bit-or-ed with suggested drive or zero for default drive.

PrefixString: FileName three character prefix (null-terminated).

Unique: Base filename numeric value or zero for system derived value.

TempFileName: Receiving pathname buffer (at least 144 bytes long).

Returns Filename's unique numeric value.

GetTextAlign

Declaration
```
function GetTextAlign(DC: HDC): Word;
```

Retrieves text-alignment flags.

Parameters *DC*: Device context identifier.

Returns Combination of text-alignment flags: *ta_Left, ta_Center, ta_Right, ta_BaseLine, ta_Bottom, ta_Top, ta_NoUpdateCP*, and *ta_UpdateCP*. See "(TA_) Text alignment options" in Chapter 1.

See also *TextOut, ExtTextOut*.

GetTextCharacterExtra

Declaration
```
function GetTextCharacterExtra(DC: HDC): Integer;
```

Retrieves extra space (in logical units) added to each character as it is written in a line.

Parameters *DC*: Device context identifier.

Returns Current inter-character spacing.

See also *TextOut, ExtTextOut.*

GetTextColor

Declaration `function GetTextColor(DC: HDC): Longint;`

Retrieves the current foreground color used to draw characters.

Parameters *DC*: Device context.

Returns RGB color value.

See also *TextOut, ExtTextOut.*

GetTextExtent

Declaration `function GetTextExtent(DC: HDC; Str: PChar; Count: Integer): Longint;`

Computes dimensions of *Str* based on the currently selected font.

Parameters *DC*: Device context identifier.

Str: Line of text.

Count: Number of characters in *Str*.

Returns Height and width (in logical units) in high and low word, respectively.

GetTextFace

Declaration `function GetTextFace(DC: HDC; Count: Integer; Facename: PChar): Integer;`

Copies selected font's typeface name to *Facename*.

Parameters *DC*: Device context identifier.

Count: Size of *Facename*.

Facename: Receiving buffer.

Returns Actual number of bytes copied.

GetTextMetrics

Declaration `function GetTextMetrics(DC: HDC; var Metrics: TTextMetric): Bool;`

Retrieves metrics of the currently selected font in *Metrics*.

Parameters *DC*: Device context identifier.

Metrics: Receiving *TTextMetric* structure.

Returns Non-zero if successful; zero if not.

GetThresholdEvent

Declaration `function GetThresholdEvent: LPInteger;`

Retrieves a recent threshold event.

Returns Pointer to threshold event.

GetThresholdStatus

Declaration `function GetThresholdStatus: Integer;`

Retrieves threshold event status where each set bit represents a voice-queue level currently below threshold.

Returns Current threshold event status flags.

GetTickCount

Declaration `function GetTickCount: Longint;`

Retrieves the elapsed time since system was started.

Returns Elapsed time (in milliseconds).

GetTopWindow

Declaration `function GetTopWindow(Wnd: HWnd): HWnd;`

Retrieves a window's top-level child window.

Parameters *Wnd*: Parent window identifier.

Returns Child window identifier; 0 if does not exist.

GetUpdateRect

Declaration `function GetUpdateRect(Wnd: HWnd; var Rect: TRect; Erase: Bool): Bool;`

Retrieves, into *ARect*, the smallest enclosing rectangle of a window's update region.

Parameters *Wnd*: Window identifier.

Rect: Receiving *TRect* structure.

Erase: Non-zero to erase update region background.

Returns Non-zero if non-empty update region; zero if otherwise.

See also *wm_EraseBkgnd*.

GetUpdateRgn

Declaration `function GetUpdateRgn(Wnd: HWnd; Rgn: HRgn; Erase: Bool): Integer;`

Copies into *Rgn* a window's update region.

Parameters *Wnd*: Window identifier.

Rgn: Receiving update region.

Erase: Non-zero if background should be erased and child windows redrawn.

Returns One of the following region types: *ComplexRegion*, *Error*, *NullRegion*, *SimpleRegion*. See "Region flags" in Chapter 1.

GetVersion

Declaration `function GetVersion: Word;`

Retrieves Window's current version number.

Returns Minor and major version number in high and low order bytes, respectively.

GetViewportExt

Declaration `function GetViewportExt(DC: HDC): Longint;`

Retrieves the viewport extents of a device context.

Parameters *DC*: Device context identifier.

Returns X- and y-extents (in device units) in low- and high-order word, respectively.

GetViewportOrg

Declaration `function GetViewportOrg(DC: HDC): Longint;`

Retrieves the viewport origin of a device context.

Parameters *DC*: Device context identifier.

Returns X- and y-coordinate (in device units) in low- and high-order word, respectively.

GetWindow

Declaration `function GetWindow(Wnd: HWnd; Cmd: Word): HWnd;`

Retrieves the window with the relationship specified in *Cmd* to the window specified in *Wnd*.

Parameters *Wnd*: Original window.

Cmd: One of the following constants: *gw_Child*, *gw_HWndFirst*, *gw_HWndLast*, *gw_HWndNext*, *gw_HWndPrev*, or *gw_Owner*. See "(GW_) Get window constants" in Chapter 1.

Returns Window identifier or 0 if not found or invalid *Cmd*.

GetWindowDC

Declaration

```
function GetWindowDC(Wnd: HWnd): HDC;
```

Retrieves a display context typically used for painting non-client areas of a window.

Parameters *Wnd*: Window identifier.

Returns Display context identifier; 0 if error.

See also *ReleaseDC*.

GetWindowExt

Declaration

```
function GetWindowExt(DC: HDC): Longint;
```

Retrieves a window's extents.

Parameters *DC*: Device context identifier.

Returns X- and y-extents (in logical units) in low- and high-order word, respectively.

GetWindowLong

Declaration

```
function GetWindowLong(Wnd: HWnd; Index: Integer): Longint;
```

Retrieves information about a window or a window's extra byte values.

Parameters *Wnd*: Window identifier.

Index: Byte offset or one of the following constants: *gwl_ExStyle*, *gwl_Style*, or *gwl_WndProc*. See "(gwl_) Window field offsets" in Chapter 1.

Returns Specified window information.

GetWindowOrg

Declaration `function GetWindowOrg(DC: HDC): Longint;`

Retrieves a window's origin.

Parameters *DC*: Device context identifier.

Returns X- and y-coordinates (in logical coordinates) in low- and high-order word.

GetWindowRect

Declaration `procedure GetWindowRect(Wnd: HWnd; var Rect: TRect);`

Retrieves the dimensions of a window's bounding rectangle into *Rect* (in screen coordinates).

Parameters *Wnd*: Window identifier.

Rect: Receiving *TRect* structure.

GetWindowsDirectory

Declaration `procedure GetWindowsDirectory(Buffer: PChar; Size: Word);`

Retrieves the Windows directory pathname into *Buffer*.

Parameters *Buffer*: Receiving pathname buffer.

Size: Size of *Buffer* (should be at least 144 bytes long).

GetWindowTask

Declaration `function GetWindowTask(Wnd: HWnd): THandle;`

Retrieves a window's application task identifier.

Parameters *Wnd*: Window identifier.

Returns Task identifier.

GetWindowText

Declaration `function GetWindowText(Wnd: HWnd; Str: PChar; MaxCount: Integer): Integer;`

Copies a window's caption or a control's text into *Str*.

Parameters *Wnd*: Window or control identifier.

Str: Receiving string buffer.

MaxCount: Size of *Str*.

Returns Actual number of bytes copied or zero if no text.

GetWindowTextLength

Declaration `function GetWindowTextLength(Wnd: HWnd): Integer;`

Retrieves a window's caption or a control's text length.

Parameters *Wnd*: Window or control identifier.

Returns Text length.

GetWindowWord

Declaration `function GetWindowWord(Wnd: HWnd; Index: Integer): Word;`

Retrieves information about a window or window-class extra byte values.

Parameters *Wnd*: Window identifier.

Index: Positive byte offset or one of the following constants: *gww_HInstance*, *gww_HWndParent*, or *gww_ID*. See "(gww_) Window field offsets" in Chapter 1.

Returns Word value.

GetWinFlags

Declaration `function GetWinFlags: Longint;`

Retrieves memory configuration flags which Windows is running.

Returns Flag mask specifying the current memory configuration. Can include: *wf_CPU286, wf_CPU386, wf_WIN286, wf_WIN386, wf_LargeFrame, wf_PMode,* and *wf_SmallFrame*. See "(wf_) Windows memory configuration flags" in Chapter 1.

GlobalAddAtom

Declaration `function GlobalAddAtom(Str: PChar): TAtom;`

Adds *Str* to the atom table, creating a new global atom.

Parameters *Str*: String (null-terminated).

Returns Newly created atom; 0 if error.

GlobalAlloc

Declaration `function GlobalAlloc(Flags: Word; Bytes: Longint): THandle;`

Allocates memory of at least the requested size from the global heap.

Parameters *Flags*: Flag mask. One or more of the following constants: *gmem_DDEShare, gmem_Discardable, gmem_Fixed, gmem_Moveable, gmem_NoCompact, gmem_NoDiscard, gmem_Not_Banked, gmem_Notify, gmem_ZeroInit*. See "(gmem_) Global memory flags" in Chapter 1.

Bytes: Number of bytes to allocate.

Returns Allocated global memory identifier; 0 if error.

See Also *GlobalSize*

GlobalCompact

Declaration `function GlobalCompact(MinFree: Longint): Longint;`

Compacts global memory and, if necessary, removes discardable segments to create a *MinFree* size block, if possible.

Parameters *MinFree*: Desired free bytes or zero to return largest free segment if all discardable segments are removed.

Returns Size of largest free block.

GlobalDeleteAtom

Declaration `function GlobalDeleteAtom(AnAtom: TAtom): TAtom;`

Decreases a global atom's reference count by one, deleting the atom and associated string from the atom table if the reference count becomes zero.

Parameters *AnAtom*: TAtom identifier.

Returns 0 if successful; *AnAtom* if not.

GlobalFindAtom

Declaration `function GlobalFindAtom(Str: PChar): TAtom;`

Retrieves the global atom associated with *Str*.

Parameters *Str*: Search string (null-terminated).

Returns Global atom; 0 if *Str* not in table.

GlobalFix

Declaration `procedure GlobalFix(Mem: THandle);`

Fixes a global memory block in memory and increases it lock reference count by one.

Parameters *Mem*: Global memory block identifier.

See also *GlobalUnfix*

GlobalFlags

Declaration `function GlobalFlags(Mem: THandle): Word;`

Retrieves information about *Mem*.

Parameters *Mem*: Global memory block identifier.

Returns *gmem_DDEShare, gmem_Discardable, gmem_Discarded,* or *gmem_Not_Banked* in high byte and lock reference-count in low byte. See "(gmem_) Global memory flags" in Chapter 1.

GlobalFree

Declaration `function GlobalFree(Mem: THandle): THandle;`

Frees a unlocked global memory block and invalids its handle.

Parameters *Mem*: Global memory block identifier.

Returns 0 if successful; *Mem* if not.

GlobalGetAtomName

Declaration `function GlobalGetAtomName(AnAtom: TAtom; Buffer: PChar; Size: Integer): Word;`

Copies the string associated with *AnAtom* into *Buffer*.

Parameters *AnAtom*: Atom identifier.

Buffer: Receive buffer.

Size: Size of *Buffer*.

Returns Actual number of bytes copied; zero if *AnAtom* not valid.

GlobalHandle

Declaration `function GlobalHandle(Mem: Word): Longint;`

Retrieves the handle of a global memory object with the specified segment address.

Parameters *Mem*: Segment address.

Returns Handle and segment address in low- and high-word, respectively; 0 if does not exist.

GlobalLock

Declaration `function GlobalLock(Mem: THandle): Pointer;`

Increments the reference count of a global memory block and returns a pointer to it.

Parameters *Mem*: Global memory block identifier.

Returns Memory block address if successful; **nil** if not.

GlobalLRUNewest

Declaration `function GlobalLRUNewest(Mem: THandle): THandle;`

Minimizes the likelihood that a global memory object will be discarded by moving it into the newest least-recently-used memory position.

Parameters *Mem*: Global memory object identifier.

Returns 0 if invalid *Mem*.

GlobalLRUOldest

Declaration `function GlobalLRUOldest(Mem: THandle): THandle;`

Maximizes the likelihood that a global memory object will be discarded by moving it into the oldest least-recently-used memory position.

Parameters *Mem*: Global memory object identifier.

Returns 0 if invalid *Mem*.

GlobalNotify

Declaration `procedure GlobalNotify(NotifyProc: TFarProc);`

Calls *NotifyProc* passing the handle of the global memory block to be discarded. If *NotifyProc* returns non-zero the block is discarded.

Parameters *NotifyProc*: Notification callback procedure's instance-address.

GlobalPageLock

Declaration `function GlobalPageLock(Selector: THandle): Word;`

Increments a memory block's page-lock count. Lock operations may be nested.

Parameters *Selector*: Memory selector.

Returns Incremented page-lock count if successful; zero if not.

See also *GlobalPageUnlock*

GlobalPageUnlock

Declaration `function GlobalPageUnlock(Selector: THandle): Word;`

Decrements a memory block's page-lock count. If count reaches zero the page is allowed to move or be swapped to disk.

Parameters *Selector*: Memory selector.

Returns Decremented page-lock count if successful; zero if not.

See also *GlobalPageLock*

GlobalReAlloc

Declaration `function GlobalReAlloc(Mem: THandle; Bytes: Longint; Flags: Word): THandle;`

Reallocates a global memory block to *Bytes* size.

Parameters *Mem*: Global memory block identifier.

Bytes: New size of *Mem*.

Flags: One or more of *gmem_Discardable, gmem_Notify, gmem_Moveable, gmem_NoCompact, gmem_NoDiscard, gmem_ZeroInit*. See "(gmem_) Global memory flags" in Chapter 1.

Returns Reallocated global memory block identifier; 0 if error.

GlobalSize

Declaration `function GlobalSize(Mem: THandle): Longint;`

Retrieves the current size of a global memory block.

Parameters *Mem*: Global memory block identifier.

Returns Actual size (in bytes); zero if *Mem* invalid or discarded.

GlobalUnfix

Declaration `function GlobalUnfix(Mem: THandle): Bool;`

Unlocks a global memory block locked by *GlobalFix*. If the block's lock reference-count reaches zero the block is subject to moving or discarding.

Parameters *Mem*: Global memory block identifier.

Returns Zero if lock reference-count decrements to zero; non-zero if not.

GlobalUnlock

Declaration `function GlobalUnlock(Mem: THandle): Bool;`

Unlocks a global memory block locked by *GlobalLock*. If the block's lock reference-count reaches zero the block is subject to moving or discarding.

Parameters *Mem*: Global memory block identifier.

Returns Zero if lock reference-count decrements to zero; non-zero if not.

GlobalUnWire

Declaration `function GlobalUnWire(Mem: THandle): Bool;`

Unlocks a memory segment locked by *GlobalWire*.

Parameters *Mem*: Segment identifier.

Returns Non-zero if segment unlocked; zero if not.

GlobalWire

Declaration `function GlobalWire(Mem: THandle): PChar;`

Moves a segment, that must be locked for a long period, into low memory and locks it.

Parameters *Mem*: Segment identifier.

Returns New segment location if successful; **nil** if not.

See also *GlobalUnWire*.

GrayString

Declaration `function GrayString(DC: HDC; Brush: HBrush; OutputFunc: TFarProc; Data: Longint; Count, X, Y, Width, Height: Integer): Bool;`

Draws gray text, using the currently selected font, by calling *OutputFunc* and passing *DC* (with a bitmap of *Height* and *Width* size), *Data*, and *Count*.

Parameters *DC*: Device context identifier.

Brush: *HBrush* used for graying.

OutputFunc: Draw function procedure-instance address or **nil** to use *TextOut*.

Data: Data to pass to *OutputFunc* or string if *OutputFunc* is 0.

Count: Size of *Data* or zero and *Data* is a string to calculate length or –1 and *OutputFunc* returns zero and image is displayed but not shown.

X, Y: Starting logical position of enclosing rectangle.

Width: Width (in logical units) of enclosing rectangle or zero and *Data* is a string to calculate width.

Height: Height (in logical units) of enclosing rectangle or zero and *Data* is a string to calculate height.

Returns Non-zero if successful; zero if output function returned zero or insufficient memory to create bitmap.

See also *GetSysColor, SetTextColor, color_Graytext, mm_Text.*

HideCaret

Declaration

```
procedure HideCaret(Wnd: HWnd);
```

Non-destructively removes the caret from the display screen.

Parameters *Wnd*: Owning window of caret or 0 if owning window is in current task.

See Also *ShowCaret*

HiliteMenuItem

Declaration

```
function HiliteMenuItem(Wnd: HWnd; Menu: HMenu; IDHilite, Hilite: Word):
Bool;
```

Highlights or removes highlighting from a top-level menu item.

Parameters *Wnd*: Window identifier.

Menu: Top-level menu identifier.

IDHilite: Integer ID or position of menu item.

Hilite: A combination of *mf_ByCommand* or *mf_ByPosition* with *mf_Hilite* or *mf_Unhilite*. See "(MF_) Menu flags" in Chapter 1.

Returns Non-zero if successful; zero if not.

HiWord

Declaration

```
function HiWord(AnInteger: Longint): Word;
```

Extracts the high-order word from a 32-bit integer value.

Parameters *AnInteger*: The 32-bit integer.

Returns The high-order word.

InflateRect

Declaration

```
procedure InflateRect(var Rect: TRect; X, Y: Integer);
```

Modifies the width and height of *Rect*. Adds *X* to left and right ends and *Y* to top and bottom of rectangle.

Parameters *Rect*: *TRect* structure.

X: Positive or negative value to change rectangle width.

Y: Positive or negative value to change rectangle height.

InitAtomTable

Declaration

```
function InitAtomTable(Size: Integer): Bool;
```

Initializes the atom hash table and sets its size (37 by default).

Parameters *Size*: Number of table entries in atom hash table (should be prime).

Returns Non-zero if successful; zero if not.

InSendMessage

Declaration `function InSendMessage: Bool;`

Determines whether the message being processed by the current window function was sent via a *SendMessage* call.

Returns Non-zero if message sent by *SendMessage*; zero if not.

InsertMenu

Declaration
```
function InsertMenu(Menu:HMenu; Position, Flags, IDNewItem: Word;
NewItem: PChar): Bool;
```

Inserts a new menu item whose state is specified by *Flags*.

Parameters *Menu*: Menu identifier.

Position: Command ID or position of menu item to insert new menu item after or –1 to append to end.

Flags: *mf_ByCommand* or *mf_ByPosition* combined with any of the following: *mf_Bitmap, mf_String, mf_OwnerDraw, mf_Separator, mf_Popup mf_MenuBarBreak, mf_MenuBreak, mf_Checked,* and *mf_Unchecked*. See "(MF_) Menu flags" in Chapter 1.

IDNewItem: New menu item command ID or menu handle if popup.

NewItem: New menu item contents.

Returns Non-zero if successful; zero if not.

See also *DrawMenuBar, wm_DrawItem, wm_MeasureItem*

IntersectClipRect

Declaration `function IntersectClipRect(DC: HDC; X1, Y1, X2, Y2: Integer): Integer;`

Creates a new clipping region from the intersection of the region and the specified rectangle.

Parameters *DC*: Device context identifier.

X1, Y1: Upper-left corner of rectangle.

X2, Y2: Lower-right corner of rectangle.

Returns One of *ComplexRegion, Error, NullRegion,* or *SimpleRegion.* See "Region flags" in Chapter 1.

IntersectRect

Declaration

```
function IntersectRect(var DestRect, Src1Rect, Src2Rect: TRect): Integer;
```

Determines the intersection of two rectangles.

Parameters *DestRect*: *TRect* structure representing the resulting rectangle.

Src1Rect: *TRect* structure representing a source rectangle.

Src2Rect: *TRect* structure representing a source rectangle.

Returns Non-zero if intersection is not empty; zero if intersection is empty.

InvalidateRect

Declaration

```
procedure InvalidateRect(Wnd: HWnd; Rect: LPRect; Erase: Bool);
```

Invalidates a windows client area by adding *Rect* to the window's update region.

Parameters *Wnd*: Window identifier.

Rect: *TRect* to be added to update region or **nil** for entire client area.

Erase: Non-zero for *BeginPaint* to erase background.

See also *ValidateRect, ValidateRgn, wm_Paint.*

InvalidateRgn

Declaration

```
procedure InvalidateRgn(Wnd: HWnd; Rgn: HRgn; Erase: Bool);
```

Invalidates a window's client area by adding *Rgn* to the window's update region.

Parameters *Wnd*: Window identifier.

Rgn: Region identifier (in client coordinates).

Erase: Non-zero for *BeginPaint* to erase background.

See also *ValidateRect, ValidateRgn, wm_Paint.*

InvertRect

Declaration `procedure InvertRect(DC: HDC; var Rect: TRect);`

Inverts the colors of the rectangle specified by *Rect*.

Parameters *DC*: Device context identifier.

Rect: *TRect* structure (in logical coordinates).

InvertRgn

Declaration `function InvertRgn(DC: HDC; Rgn: HRgn): Bool;`

Inverts the colors of the region specified by *Rgn*.

Parameters *DC*: Device context identifier.

Rgn: Region identifier (in device units).

Returns Non-zero if successful; zero if not.

IsCharAlpha

Declaration `function IsCharAlpha(AChar: Char): Bool;`

Uses language driver and current language to determine if *AChar* is alphabetical.

Parameters *AChar*: Test character.

Returns Non-zero if alphabetical; zero if not.

IsCharAlphaNumeric

Declaration `function IsCharAlphaNumeric(AChar: Char): Bool;`

Uses language driver and current language to determine if *AChar* is alphanumeric.

Parameters *AChar*: Test character.

Returns Non-zero if alphanumeric; zero if not.

IsCharLower

Declaration `function IsCharLower(AChar: Char): Bool;`

Uses language driver and current language to determine if *AChar* is lowercase.

Parameters *AChar*: Test character.

Returns Non-zero if lowercase; zero if not.

IsCharUpper

Declaration `function IsCharUpper(AChar: Char): Bool;`

Uses language driver and current language to determine if *AChar* is uppercase.

Parameters *AChar*: Test character.

Returns Non-zero if uppercase; zero if not.

IsChild

Declaration `function IsChild(Parent, Wnd: HWnd): Bool;`

Determines if *Wnd* is a child window of *Parent*.

Parameters *Parent*: Window identifier.

Wnd: Test window.

Returns Non-zero if child window; zero if not.

IsClipboardFormatAvailable

Declaration `function IsClipboardFormatAvailable(Format: Word): Bool;`

Determines whether data in the specified format exists in the clipboard.

Parameters *Format*: Registered clipboard format. One of the *cf_* constants. See "(CF_) Clipboard formats" in Chapter 1.

Returns Non-zero if data in *Format* available; zero if not.

See also *SetClipboardData*.

IsDialogMessage

Declaration `function IsDialogMessage(Dlg: HWnd; var Msg: TMsg): Bool;`

Determines and processes messages for modeless dialog boxes, converting keyboard messages into command messages.

Parameters *Dlg*: Dialog box identifier.

Msg: *TMsg* structure.

Returns Non-zero if *Msg* processed (*TranslateMessage* and *DispatchMessage* should not be called); zero if not.

IsDlgButtonChecked

Declaration `function IsDlgButtonChecked(Dlg: HWnd; IDButton: Integer): Word;`

Determines if button control is checked.

Parameters *Dlg*: Dialog box identifier.

IDButton: Button control ID.

Returns Non-zero if checked; zero if not. For three-state button, grayed (2), checked (1), or zero.

IsIconic

Declaration `function IsIconic(Wnd: HWnd): Bool;`

Determines whether a window is iconic (minimized).

Parameters *Wnd*: Window identifier.

Returns Non-zero if minimized; zero if not.

IsRectEmpty

Declaration `function IsRectEmpty(var Rect: TRect): Bool;`

Determines whether a rectangle's width and/or height are zero.

Parameters *Rect*: *TRect* structure.

Returns Non-zero if empty; zero if not.

IsWindow

Declaration `function IsWindow(Wnd: HWnd): Bool;`

Determines if *Wnd* is a valid existing window.

Parameters *Wnd*: Window identifier.

Returns Non-zero if valid; zero if not.

IsWindowEnabled

Declaration `function IsWindowEnabled(Wnd: HWnd): Bool;`

Determines whether a window is enabled for mouse and keyboard input.

Parameters *Wnd*: Window identifier.

Returns Non-zero if enabled; zero if not.

IsWindowVisible

Declaration `function IsWindowVisible(Wnd: HWnd): Bool;`

Determines whether a window has been made visible by *ShowWindow*.

Parameters *Wnd*: Window identifier.

Returns Non-zero if exist on screen (even if completely covered); zero if not.

IsZoomed

Declaration `function IsZoomed(Wnd: HWnd): Bool;`

Determines whether a window is maximized.

Parameters *Wnd*: Window identifier.

Returns Non-zero if maximized; zero if not.

KillTimer

Declaration `function KillTimer(Wnd: HWnd; IDEvent: Integer): Bool;`

Kills a timer event removing any associated *wm_Timer* messages from the message queue.

Parameters *Wnd*: Window identifier.

IDEvent: Timer event ID.

Returns Non-zero if successful; zero if invalid *IDEvent*.

See also *SetTimer*.

_lclose

Declaration `function _lclose(FileHandle: Integer): Integer;`

Closes the specified file.

Parameters *FileHandle*: DOS file handle.

Returns Zero if successful; –1 if not.

_lcreat

Declaration `function _lcreat(PathName: PChar; Attribute: Integer): Integer;`

Opens the specified file.

Parameters *PathName*: The full DOS pathname of the file to be opened.

Attribute: One of: (0) read or write, (1) read only, (2) hidden, or (3) system file.

Returns DOS file handle if successful; –1 if not.

LimitEmsPages

Declaration `procedure LimitEmsPages(Kbytes: Longint);`

Limits number of kilobytes of expanded memory that Windows assigns to an application when running under expanded memory configuration.

Parameters *Kbytes*: Number of kilobytes.

LineDDA

Declaration `procedure LineDDA(X1, Y1, X2, Y2: Integer; LineFunc: TFarProc; Data: Pointer);`

Computes all successive points in a line and calls *LineFunc* passing *X* and *Y* of the point and *Data*.

Parameters *X1, Y1*: First point in line.

X2, Y2: Last point in line.

LineFunc: Callback function procedure-instance address.

Data: Data passed to *LineFunc*.

LineTo

Declaration `function LineTo(DC: HDC; X, Y: Integer): Bool;`

Draws a line, using selected pen, from the current position up to the specified endpoint.

Parameters *DC*: Device context identifier.

X, Y: Endpoint of the line.

Returns Non-zero if drawn; zero if not.

_llseek

Declaration `function _llseek(FileHandle: Integer; Offset: Longint; Origin: Integer): Longint;`

Positions the pointer in an open file.

Parameters *FileHandle*: DOS file handle.

Offset: Number of bytes to move the pointer.

Origin: Specifies the starting point and direction of the move: (0) forward from the beginning, (1) from the current position, or (2) back from the end of the file.

Returns The new offset of the pointer, or –1 if unsuccessful.

LoadAccelerators

Declaration `function LoadAccelerators(Instance: THandle; TableName: PChar): THandle;`

Loads the named accelerator table from an executable file.

Parameters *Instance*: Module instance whose executable file contains the accelerator table.

TableName: Accelerator table name (null-terminated) or integer ID.

Returns Accelerator table identifier if successful; 0 if not.

LoadBitmap

Declaration

```
function LoadBitmap(Instance: THandle; BitmapName: PChar): HBitmap;
```

Loads the named bitmap resource.

Parameters *Instance*: Module instance whose executable file contains the bitmap or 0 for predefined bitmap.

BitmapName: String (null-terminated) or integer ID specifying the bitmap, or a predefined bitmap, specified by an *obm_* constant. See "(obm_) Predefined bitmaps" in Chapter 1.

Returns Bitmap identifier if successful; 0 if not.

LoadCursor

Declaration

```
function LoadCursor(Instance: THandle; CursorName: PChar): HCursor;
```

Loads the named cursor resource.

Parameters *Instance*: Module instance whose executable file contains the cursor or 0 for predefined cursor.

CursorName: String (null-terminated) or integer ID name or a predefined cursor, specified with one of the *idc_* constants. See "(idc_) Standard cursor IDs" in Chapter 1.

Returns Cursor identifier if successful; 0 if cursor not found; undefined if not a cursor resource.

LoadIcon

Declaration

```
function LoadIcon(Instance: THandle; IconName: PChar): HIcon;
```

Loads the named icon resource.

Parameters *Instance*: Module instance whose executable file contains the icon or 0 for predefined icon.

IconName: String or integer ID name or predefined icon, specified with one of the *idi_* constants. See "(idi_) Standard icon IDs" in Chapter 1.

Returns Icon identifier if successful; 0 if not.

LoadLibrary

Declaration
```
function LoadLibrary(LibFileName: PChar): THandle;
```

Loads the named library module.

Parameters *LibFileName*: Library filename (null-terminated).

Returns Library module instance identifier (value greater than 32) if successful; if not, its value is less than 32, and one of the following: (0) no memory, (5) attempt to link task, (6) multiple data segments, (10) invalid Windows version, (11) invalid EXE file, (12) OS/2 app, (13) DOS 4.0 app, (14) Invalid EXE type, (15) not protect mode.

LoadMenu

Declaration
```
function LoadMenu(Instance: THandle; MenuName: PChar): HMenu;
```

Loads the named menu resource.

Parameters *Instance*: Module instance whose executable file contains the menu.

MenuName: String (null-terminated) or integer ID name of menu.

Returns Menu identifier if successful; 0 if not.

LoadMenuIndirect

Declaration
```
function LoadMenuIndirect(var MenuTemplate): HMenu;
```

Loads the menu defined by *MenuTemplate*.

Parameters *MenuTemplate*: Array of *TMenuItemTemplate* structures.

Returns Menu identifier if successful; 0 if not.

LoadModule

Declaration `function LoadModule(ModuleName: PChar; ParameterBlock: Pointer): THandle;`

Loads and runs a Windows application.

Parameters *ModuleName*: Application filename (null-terminated).

ParameterBlock: Four field structure: *EnvSeg*: *Word*, environment segment address or zero for Windows environment; *CmdLine*: *Longint*, command line; *CmdShow*: *Longint*, two *Word*-length structure, first word set to 2, second set to *CmdShow* of *ShowWindow*; *Reserved*: *Longint*, must be 0.

Returns Same as *LoadLibrary*.

See also *WinExec*.

LoadResource

Declaration `function LoadResource(Instance, ResInfo: THandle): THandle;`

Allocates memory and loads a resource.

Parameters *Instance*: Module instance whose executable file contains the resource.

ResInfo: Resource identifier (returned by *FindResource*).

Returns Resource identifier if successful; 0 if not.

See also *LockResource*.

LoadString

Declaration `function LoadString(Instance: THandle; ID: Word; Buffer: PChar; BufferMax: Integer): Integer;`

Loads the named string resource and copies it into *Buffer* appending a null-character.

Parameters *Instance*: Module instance whose executable file contains the string.

ID: Integer ID of string.

Buffer: Receiving buffer.

BufferMax: Size of *Buffer*.

Returns Actual number of bytes copied; zero if does not exist.

LocalAlloc

Declaration `function LocalAlloc(Flags, Bytes: Word): THandle;`

Allocates a local memory block from the local heap. The actual size may be larger than the specified size.

Parameters *Flags*: One or more of the following constants: *lmem_Discardable, lmem_Fixed, lmem_Modify, lmem_Moveable, lmem_NoCompact, lmem_NoDiscard,* and *lmem_ZeroInit*. See "(lmem_) Local memory flags" in Chapter 1.

Bytes: Size (in bytes) of block to allocate.

Returns Local memory block identifier if successful; 0 if not.

See also *LockData*.

LocalCompact

Declaration `function LocalCompact(MinFree: Word): Word;`

Generates a free block of at least *MinFree* size. If necessary, the function will move and/or discard unlocked blocks.

Parameters *MinFree*: Number of desired free bytes or zero to return largest contiguous block.

Returns Size (in bytes) of largest block.

LocalFlags

Declaration `function LocalFlags(Mem: THandle): Word;`

Retrieves information about a memory block.

Parameters *Mem*: Local memory block identifier.

Returns *lmem_Discardable* or *lmem_Discarded* in high-byte and lock count in low-byte. See "(lmem_) Local memory flags" in Chapter 1.

LocalFree

Declaration `function LocalFree(Mem: THandle): THandle;`

Frees a local memory block and invalidates its handle.

Parameters *Mem*: Local memory block identifier.

Returns 0 if successful; *Mem* if not.

LocalHandle

Declaration `function LocalHandle(Mem: Word): THandle;`

Retrieves the handle of a local memory object at the specified address.

Parameters *Mem*: Local memory object address.

Returns Local memory object identifier.

LocalInit

Declaration `function LocalIni(Segment, Start, End: Word): Bool;`

Initializes a local heap and calls *GlobalLock* to lock the segment.

Parameters *Segment*: Segment address of local heap.

Start: Offset address to start of local heap.

End: Offset address to end of local heap.

Returns Non-zero if initialized; zero if not.

LocalLock

Declaration `function LocalLock(Mem: THandle): Pointer;`

Locks *Mem* and increments its lock count. The block cannot be moved or discarded.

Parameters *Mem*: Local memory block identifier.

Returns Pointer to block if successful; **nil** if not.

LocalReAlloc

Declaration `function LocalReAlloc(Mem: THandle; Bytes, Flags: Word): THandle;`

Changes the size or attributes, as specified by *Flags*, of a local memory block.

Parameters *Mem*: Local memory block identifier.

Bytes: New size (in bytes) of block.

Flags: One or more of the constants: *lmem_Discardable, lmem_Modify, lmem_Moveable, lmem_NoCompact, lmem_NoDiscard,* and *lmem_ZeroInit*. See "(lmem_) Local memory flags" in Chapter 1.

Returns Local memory block identifier if successful; 0 if not.

See also *LockData*.

LocalShrink

Declaration `function LocalShrink(Seg: THandle; Size: Word): Word;`

Reduces the local heap to the specified size whose value can be no less than the minimum size specified in the application's module-definition file.

Parameters *Seg*: Segment containing local heap or zero for current data segment.

Size: Desired size (in bytes).

Returns Size after shrinkage.

See also *GlobalSize*.

LocalSize

Declaration `function LocalSize(Mem: THandle): Word;`

Retrieves the current size of a local memory block.

Parameters *Mem*: Local memory block identifier.

Returns Size (in bytes) of block; 0 if invalid Mem.

LocalUnlock

Declaration `function LocalUnlock(Mem: THandle): Bool;`

Unlocks a local memory block by decrementing its lock count.

Parameters *Mem*: Local memory block identifier.

Returns Zero if lock count decremented to zero (making the block subject to moving or discarding); non-zero if not.

LockData

Declaration `function LockData(Dummy: Integer): THandle;`

Locks the current moveable data segment in memory.

Parameters *Dummy*: Not used. Set to zero.

Returns Identifier for the locked data segment if successful; zero if unsuccessful.

See also *UnlockData*

LockResource

Declaration `function LockResource(ResData: THandle): Pointer;`

Retrieves the address of a loaded resource and increments its reference-count. The resource is no longer subject to moving or discarding.

Parameters *ResData*: Resource identifier returned from *LoadResource*.

Returns Pointer to loaded resource; **nil** if not.

See also *UnlockResource*.

LockSegment

Declaration `function LockSegment(Segment: Word): THandle;`

Locks a segment (except protected-mode non-discardable segments) and increments its lock reference-count.

Parameters *Segment*: Segment address or –1 for current segment.

Returns Pointer to segment; **nil** if error or discarded.

See also *UnlockSegment*.

_lopen

Declaration

```
function _lopen(PathName: PChar; ReadWrite: Integer): Integer;
```

Opens the specified file.

Parameters *PathName*: String specifying the path and file name.

ReadWrite: Specifies the read and write access, using one of the *of_* constants, such as *of_Read*, *of_ReadWrite* or *of_Write*. See "of_ Open file constants" in Chapter 1.

Returns The DOS file handle if successful; –1 if not.

LoWord

Declaration

```
function LoWord(AnInteger: Longint): Word;
```

Extracts the low-order word from a 32-bit integer value.

Parameters *AnInteger*: The 32-bit integer.

Returns The low-order word.

LPtoDP

Declaration

```
function LPtoDP(DC: HDC; var Points; Count: Integer): Bool;
```

Converts logical points in *Points*, in the current mapping mode, to device points.

Parameters *DC*: Device context identifier.

Points: Array of *TPoint* structures.

Count: Size of *Points*.

Returns Non-zero if all points converted; zero if not.

_lread

Declaration `function _lread(FileHandle: Integer; Buffer: PChar; Bytes: Integer): Word;`

Read a number of bytes from an open file.

Parameters *FileHandle*: The DOS file handle.

Buffer: Receiving buffer.

Bytes: The number of bytes to read.

Returns The number of bytes read, if successful; –1 if not; zero if at end of file.

lstrcat

Declaration `function lstrcat(Str1, Str2: PChar): PChar;`

Concatenates *Str2* to *Str1*.

Parameters *Str1*: First string (null-terminated).

Str2: Second string (null-terminated).

Returns *Str1* if successful; zero if not.

lstrcmp

Declaration `function lstrcmp(Str1, Str2: PChar): Integer;`

Performs case-sensitive, lexicographical compare of two strings based on the currently selected language. Uppercase characters evaluate lower than lowercase characters.

Parameters *Str1*: First string (null-terminated).

Str2: Second string (null-terminated).

Returns Less than zero if *Str1* < *Str2*; zero if *Str1* = *Str2*; greater than zero if *Str1* > *Str2*.

lstrcmpi

Declaration `function lstrcmpi(Str1, Str2: PChar): Integer;`

Performs case-insensitive, lexicographical compare of two strings based on the currently selected language.

Parameters *Str1*: First string (null-terminated).

Str2: Second string (null-terminated).

Returns Less than zero if *Str1* < *Str2*; zero if *Str1* = *Str2*; greater than zero if *Str1* > *Str2*.

lstrcpy

Declaration `function lstrcpy(Str1, Str2: PChar): PChar;`

Copies (including null) *Str2* to *Str1*.

Parameters *Str1*: First string (null-terminated).

Str2: Second string (null-terminated).

Returns Pointer to *Str1* if successful; zero if not.

lstrlen

Declaration `function lstrlen(Str: PChar): Integer;`

Calculates length (not including null) of *Str*.

Parameters *Str*: String (null-terminated).

Returns Length (in bytes) of *Str*.

_lwrite

Declaration `function _lwrite(FileHandle: Integer; Buffer: PChar; Bytes: Integer): Word;`

Writes data from Buffer into the specified file.

Parameters *FileHandle*: DOS file handle.

Buffer: Holds the data to be written.

Bytes: The number of bytes to be written.

Returns The number of bytes written to the file, if successful; –1 if not.

MakeLong

Declaration `function MakeLong(Low, High: Word): Longint;`

Concatenates two word values into one unsigned long value.

Parameters *Low*: The low-order word of the new unsigned long.

High: The high-order word of the new unsigned long.

Returns The resulting unsigned *Longint*.

MakeProcInstance

Declaration `function MakeProcInstance(Proc: TFarProc; Instance: THandle): TFarProc;`

Creates a procedure-instance address for the specified exported function.

Parameters *Proc*: Exported function *TFarProc* address.

Instance: Module instance identifier.

Returns Function procedure-instance address if successful; zero if not.

MapDialogRect

Declaration `procedure MapDialogRect(Dlg: HWnd; var Rect: TRect);`

Converts dialog box units in *Rect* to screen units.

Parameters *Dlg*: Dialog box identifier.

Rect: *TRect* structure.

See also *GetDialogBaseUnits*.

MapVirtualKey

Declaration `function MapVirtualKey(Code, MapType: Word): Word;`

Maps a virtual-key or scan code for a key to its corresponding scan code, virtual-key code, or ASCII value as specified by *MapType*.

Parameters *Code*: Virtual-key or scan code for a key determined by *MapType* value.

MapType: (0) virtual-key code, (1) scan code, (2) virtual-key code.

Returns If *MapType* is zero, returns scan code; if it is 1, returns virtual-key code; if it is 2, returns unshifted ASCII value.

MessageBeep

Declaration `procedure MessageBeep(BeepType: Word);`

Causes the system speaker to beep.

Parameters *BeepType*: Set to zero.

MessageBox

Declaration `function MessageBox(Parent: HWnd; Txt, Caption: PChar; TextType: Word): Integer;`

Creates and displays a dialog box containing the specified message and caption and any predefined icons and pushbutton as specified by *TextType*.

Parameters *Parent*: Owning window of message box.

Txt: Message to display (null-terminated).

Caption: Dialog box caption (null-terminated) or **nil** for "Error".

TextType: One or a combination of *mb_* constants. See "(mb_) Message box flags" in Chapter 1.

Returns One of the following if successful: *id_Abort, id_Cancel, id_Ignore, id_No, id_OK, id_Retry,* or *id_Yes;* zero if not enough memory. See "(id_) Dialog box command IDs" in Chapter 1.

ModifyMenu

Declaration `function ModifyMenu(Menu: HMenu; Position, Flags, IDNewItem: Word; NewItem: PChar): Bool;`

Changes an existing menu item whose new state is specified by *Flags*.

Parameters *Menu*: Menu identifier.

Position: Command ID or position of menu item.

Flags: Combination of *mf_ByCommand* or *mf_ByPosition*, with *mf_Disabled, mf_Enabled, mf_Grayed, mf_Bitmap, mf_String, mf_OwnerDraw,mf_Separator, mf_Popup, mf_MenuBarBreak, mf_MenuBreak, mf_Checked,* and *mf_Unchecked*. See "(mf_) Menu flags" in Chapter 1.

IDNewItem: Command ID or menu handle, if *Flags* set to *mf_Popup*, of modified menu item.

NewItem: String (*mf_String*), HBitmap (*mf_Bitmap*), or application supplied data (*mf_OwnerDraw*).

Returns Non-zero if successful; zero if not.

MoveTo

Declaration `function MoveTo(DC: HDC; X, Y: Integer): Longint;`

Moves the current position to the specified point.

Parameters *DC*: Device context identifier.

X, Y: New position.

Returns X- and y-coordinates of previous position in low- and high-order word, respectively.

MoveWindow

Declaration `procedure MoveWindow(Wnd: HWnd; X, Y, Width, Height: Integer; Repaint: Bool);`

Sends a *wm_Size* message to a window. Width and height values passed with *wm_Size* are those of the client area.

Parameters *Wnd*: Pop-up or child window identifier.

X, Y: New upper-left corner of window.

Width: New window width.

Height: New window height.

Repaint: Non-zero to repaint window after moving.

MulDiv

Declaration `function MulDiv(Number, Numerator, Denominator: Integer): Integer;`

Multiplies *Numerator* by *Number* and divides the result by *Denominator*, rounding to the nearest integer.

Parameters *Number*: A number.

Numerator: Another number.

Denominator: Another number.

Returns Result value; 32,767 or –32767 if overflow or *Denominator* is zero.

OemKeyScan

Declaration `function OemKeyScan(OemChar: Word): Longint;`

Maps *OemChar* into OEM scan codes.

Parameters *OemChar*: OEM ASCII character code (0 – $0FF).

Returns Scan code and shift state (bit 2 ctrl key, bit 1 shift key pressed) in low- and high-order word, respectively, if successful; –1 in low- and high-order word if not.

OemToAnsi

Declaration `function OemToAnsi(OemStr, AnsiStr: PChar): Bool;`

Translates *OemStr* to ANSI character set.

Parameters *OemStr*: String of OEM characters (null-terminated).

AnsiStr: Receive buffer or *OemStr* to translate in place.

Returns Always *False* (-1).

OemToAnsiBuff

Declaration `procedure OemToAnsiBuff(OemStr, AnsiStr: PChar; Length: Integer);`

Translates *OemStr* to ANSI character set.

Parameters *OemStr*: Buffer of OEM characters.

AnsiStr: Receive buffer or *OemStr* to translate in place.

Length: Size of *OemStr*.

OffsetClipRgn

Declaration `function OffsetClipRgn(DC: HDC; X, Y: Integer): Integer;`

Moves the device's clipping region according to the specified *X* and *Y* offsets.

Parameters *DC*: Device context identifier.

X: Logical units to move left or right.

Y: Logical units to move up or down.

Returns One of *ComplexRegion, Error, NullRegion,* or *SimpleRegion.* See "Region flags" in Chapter 1.

OffsetRect

Declaration

```
procedure OffsetRect(var Rect: TRect; X, Y: Integer);
```

Adjusts a rectangle's coordinates by the specified *X* and *Y* offsets.

Parameters *Rect*: *TRect* structure.

X: Units to move left or right.

Y: Units to move up or down.

OffsetRgn

Declaration

```
function OffsetRgn(Rgn: HRgn; X, Y: Integer): Integer;
```

Moves a region by the specified *X* and *Y* offsets.

Parameters *Rgn*: Region identifier.

X: Units to move left or right.

Y: Units to move up or down.

Returns One of *ComplexRegion, Error, NullRegion,* or *SimpleRegion.* See "Region flags" in Chapter 1.

OffsetViewportOrg

Declaration

```
function OffsetViewportOrg(DC: HDC; X, Y: Integer): Longint;
```

Modifies a viewport origin by summing the current origin with the specified *X* and *Y* values.

Parameters *DC*: Device context identifier.

X: X-coordinate origin offset.

Y: Y-coordinate origin offset.

Returns Y- and x-coordinates of previous origin in high- and low-order word, respectively.

OffsetWindowOrg

Declaration `function OffsetWindowOrg(DC: HDC; X, Y: Integer): Longint;`

Modifies a windows origin by summing the current origin with the specified *X* and *Y* values.

Parameters *DC*: Device context identifier.

X: X-coordinate (in logical units) origin offset.

Y: Y-coordinate (in logical units) origin offset.

Returns Y- and x-coordinates of previous origin in high- and low-order word, respectively.

OpenClipboard

Declaration `function OpenClipboard(Wnd: HWnd): Bool;`

Opens the clipboard for exclusive use by the application.

Parameters *Wnd*: Window identifier.

Returns Non-zero if successful; zero if already opened by another application.

See also *CloseClipboard*.

OpenComm

Declaration `function OpenComm(ComName: PChar; InQueue, OutQueue: Word): Integer;`

Opens a communications device. The device is initialized to a default configuration and transmit and receive queues are allocated.

Parameters *ComName*: String containing 'COMn' or 'LPTn', where n is an integer.

InQueue: Receive queue size or ignored for LPT ports.

OutQueue: Transmit queue size or ignored for LPT ports.

Returns Cid handle if successful; negative error value if not, one of: *ie_BadID, ie_BaudRate, ie_ByteSize, ie_Default, ie_Hardware, ie_Memory, ie_NOpen, ie_Open*. See "(ie_) Open comm error flags" in Chapter 1.

See also *SetCommState*.

OpenFile

Declaration

```
function OpenFile(FileName: PChar; var ReOpenBuff: TOFStruct; Style:
Word): Integer;
```

Creates, opens, reopens or deletes file.

Parameters *FileName*: The specified file name.

ReOpenBuff: Receives information about file when opened.

Style: Specifies the action. One of the *of_* constants. See "(of_) Open file constant" in Chapter 1.

Returns DOS file handle if successful; –1 if not.

OpenIcon

Declaration

```
function OpenIcon(Wnd: HWnd): Bool;
```

Restores a minimized window to its original size and position.

Parameters *Wnd*: Window identifier.

Returns Non-zero if successful; zero if not.

OpenSound

Declaration

```
function OpenSound: Integer;
```

Opens the play device for exclusive use by the application.

Returns Number of available voices; *s_serDVNA* if in use; or *s_serOFM* if insufficient memory. See "(s_) Sound constants" in Chapter 1.

OutputDebugString

Declaration `procedure OutputDebugString(OutputString: PChar);`

Sends *OutputString* to debugger if present, or otherwise to AUX port in debug version of Windows.

Parameters *OutputString*: String (null-terminated).

PaintRgn

Declaration `function PaintRgn(DC: HDC; Rgn: HRgn): Bool;`

Fills a region with the selected brush.

Parameters *DC*: Device context.

Rgn: Fill region.

Returns Non-zero if successful; zero if not.

PaletteRGB

Declaration `function PaletteRGB(Red, Green, Blue: Byte): Longint;`

Produces a palette-relative RGB color value from three primary color values ranging from 0 to 255. The return value has 2 in the high-order byte.

Parameters *Red*: The red intensity value.

Green: The green intensity value.

Blue: The blue intensity value.

Returns The resulting RGB color.

PatBlt

Declaration `function PatBlt(DC: HDC; X, Y, Width, Height: Integer; Rop: Longint): Bool;`

Creates a bit pattern by performing *Rop* using the selected brush and the pattern already on the device.

Parameters *DC*: Device context identifier.

X, Y: Upper left corner of rectangle.

Width: Width (in logical units) of rectangle.

Height: Height (in logical units) of rectangle.

Rop: One of the following raster operation codes: *PatCopy, PatInvert, DSTInvert, Blackness*, or *Whiteness*. See "Ternary raster operations" in Chapter 1.

Returns Non-zero if bit pattern drawn; zero if not.

PeekMessage

Declaration `function PeekMessage(var Msg: TMsg; Wnd: HWnd; MsgFilterMin, MsgFilterMax, RemoveMsg: Word): Bool;`

Checks application queue for a message and copies it to *Msg*. The function returns immediately if there are no messages and yields control to Windows.

Parameters *Msg*: Receiving *TMsg* structure.

Wnd: Messages destination window or 0 for any window in application or –1 for messages posted by *PostAppMessage* function.

MsgFilterMin: Lowest message ID to examine or zero for no limit.

MsgFilterMax: Highest message ID to examine or zero for no limit.

RemoveMsg: One or more of *pm_NoRemove, pm_NoYield*, and *pm_Remove*. See "(pm_) Peek message options" in Chapter 1.

Returns Non-zero if message available; zero if not.

See also *GetMessage, WaitMessage*

Pie

Declaration `function Pie(DC: HDC; X1, Y1, X2, Y2, X3, Y3, X4, Y4: Integer): Bool;`

Draws a pie-shaped wedge using the selected pen and filled with the selected brush, centered in the bounding rectangle.

Parameters *DC*: Device context identifier.

X1, Y1: Upper-left corner of bounding rectangle.

X2, Y2: Lower-left corner of bounding rectangle.

X3, Y3: Starting point of arc.

X4, Y4: Endpoint of arc.

Returns Non-zero if drawn; zero if not.

PlayMetaFile

Declaration `function PlayMetaFile(DC: HDC; MF: THandle): Bool;`

Executes the contents of a metafile on the specified device.

Parameters *DC*: Device context identifier.

MF: Metafile identifier.

Returns Non-zero if successful; zero if not.

PlayMetaFileRecord

Declaration `procedure PlayMetaFileRecord(DC: HDC; var HandleTable: THandleTable; var MetaRecord: TMetaRecord; Handles: Word);`

Executes the GDI function call contained in a metafile record.

Parameters *DC*: Device context identifier.

HandleTable: *THandleTable* used for metafile playback.

MetaRecord: *TMetaRecord* of metafile to play.

Handles: Size of *HandleTable*.

See also *EnumMetaFile*.

Polygon

Declaration `function Polygon(DC: HDC; var Points; Count: Integer): Bool;`

Draws a polygon using the current polygon-filling mode, whose vertices a specified by *Points*. The polygon is closed if necessary.

Parameters *DC*: Device context identifier.

Points: Array of *TPoint* structures.

Count: Size of *Points*.

Returns Non-zero if successful; zero if not.

See also *SetPolyFillMode*.

Polyline

Declaration `function Polyline(DC: HDC; var Points; Count: Integer): Bool;`

Draws a set of line segments using the selected pen, where each subsequent point is specified by *Points*.

Parameters *DC*: Device context identifier.

Points: Array of *TPoint* structures.

Count: Size of *Points*.

Returns Non-zero if successful; zero if not.

PolyPolygon

Declaration `function PolyPolygon(DC: HDC; var Points; var PolyCounts; Count: Integer): Bool;`

Draws a series of possibly overlapping polygons using the current polygon-filling mode, whose vertices are specified by *Points*. The polygons are not automatically closed.

Parameters *DC*: Device context identifier.

Points: Array of *TPoint* structures.

PolyCounts: Array of integers where each number specifies the number of vertices for each polygon in *Points*.

Count: Size of *PolyCounts*.

Returns Non-zero if drawn; zero if not.

PostAppMessage

Declaration

```
function PostAppMessage(Task: THandle; Msg, wParam: Word; lParam:
Longint): Bool;
```

Posts a message to an application. The *Wnd* of the message is set to 0.

Parameters *Task*: Application to receive message.

Msg: Message type.

wParam: Additional message information.

lParam: Additional message information.

Returns Non-zero if successful; zero if not.

See also *GetCurrentTask, GetMessage, PeekMessage.*

PostMessage

Declaration

```
function PostMessage(Wnd: HWnd; Msg, wParam: Word; lParam: Longint):
Bool;
```

Posts a message to an application window.

Parameters *Wnd*: Window to receive message or $FFFF for all overlapped or popup windows.

Msg: Message type.

wParam: Additional message information.

lParam: Additional message information.

Returns Non-zero if successful; zero if not.

PostQuitMessage

Declaration `procedure PostQuitMessage(ExitCode: Integer);`

Posts a *wm_Quit* message typically in response to a *wm_Destroy* message.

Parameters *ExitCode*: Application exit code (*wParam* of *wm_Quit* message).

PtInRect

Declaration `function PtInRect(var Rect: TRect; Point: TPoint): Bool;`

Determines whether a point lies within, or on the left or top side, of the specified rectangle.

Parameters *Rect*: *TRect* structure.

Point: *TPoint* structure.

Returns Non-zero if *Point* lies within *Rect*; zero if not.

PtInRegion

Declaration `function PtInRegion(Rgn: HRgn; X, Y: Integer): Bool;`

Determines whether a point lies within the specified region.

Parameters *Rgn*: Region identifier.

X, Y: A point.

Returns Non-zero if the point lies within *Rgn*; zero if not.

PtVisible

Declaration `function PtVisible(DC: HDC; X, Y: Integer): Bool;`

Determines whether a point lies within the specified device's clipping region.

Parameters *DC*: Device context identifier.

X, Y: A point.

Returns Non-zero if the point lies within the clipping region of *DC*; zero if not.

ReadComm

Declaration
```
function ReadComm(Cid: Integer; Buf: PChar, Size: Integer): Integer;
```

Reads *Cid* copying up to *Size* characters into *Buf*.

Parameters *Cid*: Communications device.

Buf: Receiving buffer.

Size: Size of *Buf*.

Returns Actual number of characters read; zero for no characters in receive queue; negative value if error.

See also *GetCommError, OpenComm.*

RealizePalette

Declaration
```
function RealizePalette(DC: HDC): Word;
```

Maps the selected logical palette to entries in the system palette.

Parameters *DC*: Device context identifier.

Returns Number of entries in logical palette mapped to different entries system palette since last realized.

Rectangle

Declaration
```
function Rectangle(DC: HDC; X1, Y1, X2, Y2: Integer): Bool;
```

Draws a rectangle using the currently selected pen and fills its interior with the currently selected brush.

Parameters *DC*: Device context identifier.

X1, Y1: Upper-left corner of rectangle.

X2, Y2: Lower-right corner of rectangle.

Returns Non-zero if rectangle drawn; zero if not.

RectInRegion

Declaration `function RectInRegion(Region: HRgn; var Rect: TRect): Bool;`

Determines whether any part of *Rect* lies within the specified region.

Parameters *Region*: *HRgn* region identifier.

Rect: *TRect* structure.

Returns Non-zero if within region boundaries; zero if not.

RectVisible

Declaration `function RectVisible(DC: HDC; var Rect: TRect): Bool;`

Determines whether any part of *Rect* lies within the specified device context clipping region.

Parameters *DC*: Device context identifier.

Rect: *TRect* structure.

Returns Non-zero if within the clipping region; zero if not.

RegisterClass

Declaration `function RegisterClass(var WndClass: TWndClass): Bool;`

Registers a window class whose attributes are specified by *WndCLass*, for subsequent use. A window class can only be registered once.

Parameters *WndClass*: *TWndClass* structure.

Returns Non-zero if class registered; zero if not.

RegisterClipboardFormat

Declaration `function RegisterClipboardFormat(FormatName: PChar): Word;`

Registers a clipboard format, incrementing the format's reference count if it had been previously registered.

Parameters *FormatName*: Format name (null-terminated).

Returns Registered format identifier ($C000 to $FFFF) if successful; zero if not.

RegisterWindowMessage

Declaration `function RegisterWindowMessage(Str: PChar): Word;`

Defines a unique system-wide window message.

Parameters *Str*: Register string (null-terminated).

Returns Message identifier ($C000 to $FFFF) if successful; zero if not.

ReleaseCapture

Declaration `procedure ReleaseCapture;`

Releases mouse capture restoring normal input processing.

See also *SetCapture.*

ReleaseDC

Declaration `function ReleaseDC(Wnd: HWnd; DC: HDC): Integer;`

Releases a common or window (has no affect on class or private) device context making it available to other applications.

Parameters *Wnd*: Window identifier.

DC: Device context identifier.

Returns One if device context released; zero if not.

See also *GetDC, GetWindowDC.*

RemoveFontResource

Declaration `function RemoveFontResourc(FileName: PChar): Bool;`

Removes a font from Windows font table. The font is not removed until all outstanding references to the resource are deleted.

Parameters *FileName*: Font-resource filename (null terminated) or module instance handle.

Returns Non-zero if successful; zero if not.

See also *DeleteObject, wm_FontChange.*

RemoveMenu

Declaration `function RemoveMenu(Menu: HMenu; Position, Flags: Word): Bool;`

Non-destructively removes a menu item and associated popup from the specified menu. The popup may be reused in subsequent operations.

Parameters *Menu*: Menu identifier.

Position: Command ID or position of menu item.

Flags: One of *mf_ByCommand* or *mf_ByPosition* specifying the nature of the Position argument. See "(MF_) Menu flags" in Chapter 1.

Returns Non-zero if successful, zero if not.

See also *DrawMenuBar, GetSubMenu.*

RemoveProp

Declaration `function RemoveProp(Wnd: HWnd; Str: PChar): THandle;`

Removes an entry specified by *Str*, from a window's property list. It is the applications responsibility to free the returned data handle.

Parameters *Wnd*: Window identifier.

Str: String (null-terminated) or an atom.

Returns Data handle to string; 0 if string not found.

See also *AddAtom.*

ReplyMessage

Declaration `procedure ReplyMessage(Reply: Longint);`

Replies to a message sent by a *SendMessage* call allowing both the caller of the *SendMessage* and *ReplyMessage* to continue executing.

Parameters *Reply*: Message dependent return result.

ResizePalette

Declaration `function ResizePalette(Palette: HPalette; NumEntries: Word): Bool;`

Changes the size of a logical palette to *NumEntries*. If enlarged, the additional entries are set to black.

Parameters *Palette*: Logical palette identifier.

NumEntries: New size of palette.

Returns Non-zero if successful; zero if not.

RestoreDC

Declaration `function RestoreDC(DC: HDC; SaveDC: Integer): Bool;`

Restores a device context to the previous state specified by *SaveDC* from the context stack. State information is deleted if *SaveDC* is not at top of stack.

Parameters *DC*: Device context identifier.

SaveDC: Return value of a previous *SaveDC* call or –1 for most recently saved device context.

Returns Non-zero if restored; zero if not.

RGB

Declaration `function RGB(Red, Green, Blue: Byte): Longint;`

Produces an RGB color value from three primary color values ranging from 0 to 255.

Parameters *Red*: The red intensity value.

Green: The green intensity value.

Blue: The blue intensity value.

Returns The resulting RGB color.

RoundRect

Declaration `function RoundRect(DC: HDC; X1, Y1, X2, Y2, X3, Y3: Integer): Bool;`

Draws a rectangle with rounded corners using the selected pen and filled using the selected brush.

Parameters *DC*: Device context identifier.

X1, Y1: Upper-left corner of rectangle.

X2, Y2: Lower-right corner of rectangle.

X3: Ellipse width used to draw rounded corners.

Y3: Ellipse height used to draw rounded corners.

Returns Non-zero if drawn; zero if not.

SaveDC

Declaration `function SaveDC(DC: HDC): Integer;`

Saves the current state of *DC* on the context stack.

Parameters *DC*: Device context identifier.

Returns Saved device context if successful; zero if not.

See also *RestoreDC*.

ScaleViewportExt

Declaration `function ScaleViewportExt(DC: HDC; Xnum, Xdenom, Ynum, Ydenom: Integer): Longint;`

Modifies the current viewport extents.

Parameters *DC*: Device context identifier.

Xnum: Value to multiply current x-extent.

Xdenom: Value to divide current x-extent.

Ynum: Value to multiply current y-extent.

Ydenom: Value to divide current y-extent.

Returns Previous y- and x-extent in high- and low-order word, respectively.

ScaleWindowExt

Declaration `function ScaleWindowExt(DC: HDC; Xnum, Xdenom, Ynum, Ydenom: Integer): Longint;`

Modifies the current window extents.

Parameters *Xnum*: Value to multiply current x-extent.

Xdenom: Value to divide current x-extent.

Ynum: Value to multiply current y-extent.

Ydenom: Value to divide current y-extent.

Returns Previous y- and x-extent in high- and low-order word, respectively.

ScreenToClient

Declaration `procedure ScreenToClient(Wnd: HWnd; var Point);`

Converts and replaces the screen coordinate values in *Point* with the client coordinates of the specified window.

Parameters *Wnd*: Window identifier.

Point: *TPoint* structure.

ScrollDC

Declaration `function ScrollDC(DC: HDC; dx, dy: Integer; var Scroll, Clip: TRect; UpdateRgn: HRgn; UpdateRect: LPRect): Bool;`

Scrolls a rectangle of bits by *dx* and *dy* units.

Parameters *DC*: Device context identifier.

dx: Horizontal scroll units.

dy: Vertical scroll units.

Scroll: *TRect* containing scrolling rectangle.

Clip: *TRect* containing clipping rectangle.

UpdateRgn: *ScrollDC* region uncovered by the scrolling process. If **nil**, update region is not computed.

UpdateRect: Receiving *TRect* containing the bounding rectangle of the scroll update region. If **nil**, update rectangle is not computed.

Returns Non-zero if successful; zero if not.

ScrollWindow

Declaration `procedure ScrollWindow(Wnd: HWnd; XAmount, YAmount: Integer; Rect, ClipRect: LPRect);`

Scrolls a window's client area by *XAmount* and *YAmount*. The uncovered area is combined with the window's update region.

Parameters *Wnd*: Window identifier.

XAmount: Device units to scroll in x-direction.

YAmount: Device units to scroll in y-direction.

Rect: Client area *TRect* to be scrolled or **nil** for entire client area.

ClipRect: Clipping *TRect* to be scrolled or **nil** for entire window.

See also *UpdateWindow, wm_Paint.*

SelectClipRgn

Declaration `function SelectClipRgn(DC: HDC; Rgn: HRgn): Integer;`

Uses a copy of *Rgn* as the current clipping region for the device context.

Parameters *DC*: Device context identifier.

Rgn: Region to be selected.

Returns One of the region type constants, *ComplexRegion, Error, NullRegion,* or *SimpleRegion*. See "Region flags" in Chapter 1.

SelectObject

Declaration `function SelectObject(DC: HDC; hObject: THandle): THandle;`

Selects a logical object for *DC*. Only one object can be selected at any one time and should be deleted when no longer being used.

Parameters *DC*: Device context identifier.

hObject: One of: bitmap, brush, font, pen, or region.

Returns Object being replaced or non-zero if metafile *DC*; 0 or zero if error.

See also *DeleteObject, SelectClipRgn, SelectPalette.*

SelectPalette

Declaration `function SelectPalette(DC: HDC; Palette: HPalette; ForceBackground: Bool): HPalette;`

Selects *Palette* as the device context's selected palette object which is used by GDI to control displayed colors.

Parameters *DC*: Device context identifier.

Palette: Logical palette to select.

ForceBackground: If non-zero then background palette or if zero foreground palette when window has input focus.

Returns Replaced logical palette if successful; 0 if not.

See also *CreatePalette.*

SendDlgItemMessage

Declaration `function SendDlgItemMessage(Dlg: HWnd; IDDlgItem: Integer; Msg, wParam: Word; lParam: Longint): Longint;`

Sends a message to a dialog box control specified by *IDDlgItem*. The function returns after the message is processed.

Parameters *Dlg*: Dialog box identifier.

IDDlgItem: Integer ID of destination dialog item.

Msg: Message type.

wParam: Additional message information.

lParam: Additional message information.

Returns Value returned by control's window function; zero if invalid *IDDlgItem*.

SendMessage

Declaration `function SendMessage(Wnd: HWnd; Msg, wParam: Word; lParam: Longint): Longint;`

Sends a message to the window function of the specified window. The function does not return until the message is processed.

Parameters *Wnd*: Window to receive message or $FFFF to send to all popup windows in system.

Msg: Message type.

wParam: Additional message information.

lParam: Additional message information.

Returns Value returned by receiving window function.

SetActiveWindow

Declaration `function SetActiveWindow(Wnd: HWnd): HWnd;`

Activates a top-level window.

Parameters *Wnd*: Window identifier.

Returns Previously active window.

SetBitmapBits

Declaration `function SetBitmapBits(Bitmap: HBitmap; Count: Longint; Bits: Pointer): Longint;`

Sets the bits of *Bitmap* to the bit values in *Bits*.

Parameters *Bitmap*: *HBitmap* to set.

Count: Size (in bytes) of *Bits*.

Bits: *Byte* array of bitmap bits.

Returns Number of bytes used to set bitmap bits; zero if error.

SetBitmapDimension

Declaration `function SetBitmapDimension(ABitmap: HBitmap; X, Y: Integer): Longint;`

Sets a bitmap's height and width in 0.1-millimeter units.

Parameters *Bitmap*: Bitmap identifier.

X: Bitmap width (in 0.1-millimeter units).

Y: Bitmap height (in 0.1-millimeter units).

Returns Height and width of previous dimensions in high- and low-order word, respectively.

See also *GetBitmapDimension*.

SetBkColor

Declaration `function SetBkColor(DC: HDC; Color: TColorRef): Longint;`

Sets the current background to *Color* or the nearest physical color supported by the device.

Parameters *DC*: Device context identifier.

Color: New background *TColorRef*.

Returns Previous RGB background color if successful; $80000000 if not.

See also *BitBlt*, *SetBkMode*, *StretchBlt*.

SetBkMode

Declaration `function SetBkMode(DC: HDC; BkMode: Integer): Integer;`

Sets the background mode which specifies whether or not GDI should remove existing background colors before drawing text, hatched brushes, or using non-solid pen styles.

Parameters *DC*: Device context identifier.

BkMode: One of the two modes *Opaque* or *Transparent*. See "Background modes" in Chapter 1.

Returns Previous mode if successful; zero if not.

SetBrushOrg

Declaration `function SetBrushOrg(DC: HDC; X, Y: Integer): Longint;`

Sets the origin of the selected brush. The brush should not be a stock object.

Parameters *DC*: Device context identifier.

X, *Y*: New origin (in device units), ranging from 0 to 7.

Returns Previous origin where x- and y-coordinate are in low- and high-order word, respectively.

SetCapture

Declaration `function SetCapture(Wnd: HWnd): HWnd;`

Causes all cursor input to be sent to *Wnd*, regardless of the position of the mouse.

Parameters *Wnd*: Window identifier.

Returns Previous window which received all mouse input; 0 if no such window.

See also *ReleaseCapture*.

SetCaretBlinkTime

Declaration `procedure SetCaretBlinkTime(MSeconds: Word);`

Sets the elapsed time between caret flashes.

Parameters *MSeconds*: Blink rate (in milliseconds).

SetCaretPos

Declaration `procedure SetCaretPos(X, Y: Integer);`

Moves the caret to the specified position.

Parameters *X, Y*: New position (in logical coordinates).

SetClassLong

Declaration `function SetClassLong(Wnd: HWnd; Index: Integer; NewLong: Longint): Longint;`

Replaces the long value specified by *Index* in the window's *TWndClass* structure.

Parameters *Wnd*: Window identifier.

Index: *gcl_MenuName, gcl_WndProc* or positive byte offset to set extra four-byte values. See "(gcl_) Class field offsets" Chapter 1.

NewLong: Replacement value.

Returns Previous value.

SetClassWord

Declaration `function SetClassWord(Wnd: HWnd; Index: Integer; NewWord: Word): Word;`

Replaces the word value specified by *Index* in the window's *TWndClass* structure.

Parameters *Wnd*: Window identifier.

Index: One of *gcw_CBClsExtra, gcw_CBWndExtra, gcw_HBrBackground, gcw_HCursor, gcw_HIcon, gcw_HModule, gcw_Style,* or positive byte offset to set extra two-byte value. See "(gcw_) Class field offsets" Chapter 1.

NewWord: Replacement value.

Returns Previous value.

SetClipboardData

Declaration `function SetClipboardData(Format: Word; Mem: THandle): THandle;`

Sets a data handle to the clipboard in *Format*. In most cases the data handle is freed before the function returns.

Parameters *Format*: One of the *cf_* clipboard format constants. See "(CF_) Clipboard formats" in Chapter 1.

Mem: Handle to global memory block containing data in *Format* or 0 for *wm_RenderFormat* message.

Returns Data identifier assigned by clipboard.

SetClipboardViewer

Declaration `function SetClipboardViewer(Wnd: HWnd): HWnd;`

Adds a window to the chain of windows to be notified by a *wm_DrawClipboard* message whenever the clipboard changes.

Parameters *Wnd*: Window identifier.

Returns Next window in clipboard-viewer chain.

See also *ChangeClipboardChain, wm_ChangeCBChain, wm_DrawClipboard, wm_Destroy.*

SetCommBreak

Declaration `function SetCommBreak(Cid: Integer): Integer;`

Suspends character transmission and places the device's transmission line in a break state.

Parameters *Cid*: Communications device.

Returns Zero if successful; negative if invalid *Cid*.

See also *ClearCommBreak, OpenComm.*

SetCommEventMask

Declaration `function SetCommEventMask(Cid: Integer; EvtMask: Word): PWord;`

Enables and retrieves the current state of a device's event mask.

Parameters *Cid*: Communications device.

EvtMask: Any combination of *ev_Break, ev_CTS, ev_DSR, ev_Err, ev_PErr, ev_Ring, ev_Rlsd, ev_RxChar, ev_RxFlag,* and *ev_TxEmpty*. See "(EV_) Comm event constants" in Chapter 1.

Returns Pointer to an event mask, each bit set indicates an occurrence of the event.

See also *OpenComm.*

SetCommState

Declaration `function SetCommState(var DCB: TDCB): Integer;`

Re-initializes a communications device specified by the *Id* field of *DCB*, to the state specified by *DCB*. Transmit and receive queues are not affected.

Parameters *DCB*: *TDCB* structure.

Returns Zero if successful; negative if not.

SetCursor

Declaration `function SetCursor(Cursor: HCursor): HCursor;`

Sets the cursor shape to the specified cursor resource.

Parameters *Cursor*: Cursor resource identifier (previously returned by *LoadCursor*).

Returns Previous cursor shape; 0 if no previous cursor.

SetCursorPos

Declaration `procedure SetCursorPos(X, Y: Integer);`

Moves the cursor to the specified screen coordinates. The position is adjusted if lies outside the *ClipCursor* rectangle.

Parameters *X, Y*: New position (in screen coordinates) of the cursor.

SetDIBits

Declaration
```
function SetDIBits(DC: HDC; Bitmap: THandle; StartScan, NumScans: Word;
Bits: Pointer; var BitsInfo: TBitmapInfo; Usage: Word): Integer;
```

Sets a bitmap's bits to the given values of a device-independent bitmap (DIB) specification.

Parameters *DC*: Device context identifier.

Bitmap: Bitmap identifier.

StartScan: Scan line number corresponding to first scan line in *Bits*.

NumScans: Number of scan lines in *Bits*.

Bits: Array of bytes containing the DIB bits whose format is specified by the *biBitCount* field of *BitsInfo*.

BitsInfo: *TBitmapInfo* containing DIB information.

Usage: Describes contents of *bmiColors* fields of *BitsInfo;* either *DIB_RGB_Colors* or *DIB_Pal_Colors*. See "(DIB_) Color table identifiers" in Chapter 1.

Returns Actual number of scan lines copied if successful; zero if not.

SetDIBitsToDevice

Declaration

```
function SetDIBitsToDevice(DC: HDC; DestX, DestY, Width, Height, SrcX,
SrcY, StartScan, NumScans: Word; Bits: Pointer; var BitsInfo:
TBitmapInfo; Usage: Word): Integer;
```

Sets the bits on a device surface directly from a device-independent bitmap (DIB).

Parameters

DC: Device context identifier.

DestX, DestY: Device destination rectangle origin.

Width: DIB rectangle x-extent.

Height: DIB rectangle y-extent.

SrcX, SrcY: DIB source position.

StartScan: DIB Scan line number corresponding to first scan line in *Bits*.

NumScans: DIB scan lines in *Bits*.

Bits: Array of bytes containing the DIB bits whose format is specified by the *biBitCount* field of *BitsInfo*.

BitsInfo: *TBitmapInfo* containing DIB information.

Usage: Describes contents of *bmiColors* fields of *BitsInfo;* either *DIB_RGB_Colors* or *DIB_Pal_Colors*. See "(DIB_) Color table identifiers" in Chapter 1.

Returns Number of set scan lines.

SetDlgItemInt

Declaration

```
procedure SetDlgItemInt(Dlg: HWnd; IDDlgItem: Integer; Value: Word;
Signed: Bool);
```

Sets the text of a dialog box control to the converted string value specified by *Value*.

Parameters

Dlg: Dialog box identifier.

IDDlgItem: Control's integer ID.

Value: Set value.

Signed: Non-zero if *Value* is signed.

See also *wm_SetText*.

SetDlgItemText

Declaration `procedure SetDlgItemText(Dlg: HWnd; IDDlgItem: Integer; Str: PChar);`

Sets the caption or text of a dialog box control to the value specified by *Str*.

Parameters *Dlg*: Dialog box identifier.

IDDlgItem: Control's integer ID.

Str: String (null-terminated).

See also *wm_SetText*.

SetDoubleClickTime

Declaration `procedure SetDoubleClickTime(Count: Word);`

Sets maximum elapsed time allowed between the first and second clicks of a mouse double click.

Parameters *Count*: Milliseconds between double clicks or zero to use default (500) value.

SetEnvironment

Declaration `function SetEnvironment(PortName, Environ: PChar; Count: Word): Integer;`

Creates or replaces a device's environment.

Parameters *PortName*: System port name (null-terminated).

Environ: Buffer containing new environment.

Count: Number of bytes in *Environ* to copy or zero to delete current environment.

Returns Actual number of bytes copied; zero if error; –1 if environment is deleted.

SetErrorMode

Declaration `function SetErrorMode(Mode: Word): Bool;`

Specifies whether Windows displays an error box on DOS INT 24H errors. If not, Windows fails the original INT 21H call allowing the application to handle the error.

Parameters *Mode*: (0) Windows displays error box, (1) Windows fails to the application.

Returns Non-zero if set; zero if not.

SetFocus

Declaration `function SetFocus(Wnd: HWnd): HWnd;`

Assigns input focus to a window directing all keyboard input to the window.

Parameters *Wnd*: Window identifier or 0 to ignore key strokes.

Returns Previous window having input focus; 0 if no such window.

SetHandleCount

Declaration `function SetHandleCount(Number: Word): Word;`

Changes the number of file handles available to task to the value specified by *Number*.

Parameters *Number*: Number of required file handles (maximum of 255).

Returns Actual number of file handles made available (may be less than *Number*).

SetKeyboardState

Declaration `procedure SetKeyboardState(var KeyState: Byte);`

Copies *KeyState* into Windows keyboard-state table.

Parameters *KeyState*: Array of 255 bytes containing key states.

See also *GetKeyboardState*.

SetMapMode

Declaration `function SetMapMode(DC: HDC; MapMode: Integer): Integer;`

Sets the device context mapping mode which defines logical to device units transformations for GDI and x- and y-axis orientation.

Parameters *DC*: Device context identifier.

MapMode: One of these mapping mode constants: *mm_Anisotropic, mm_HiEnglish, mm_HiMetric, mm_Isotropic, mm_LoEnglish, mm_LoMetric, mm_Text,* or *mm_Twips*. See "(mm_) Mapping modes" in Chapter 1.

Returns Previous mapping mode.

SetMapperFlags

Declaration `function SetMapperFlags(DC: HDC; Flag: Longint): Longint;`

Alters the font-mapper algorithm as specified by *Flag*, for mapping logical to physical fonts.

Parameters *DC*: Device context identifier.

Flag: If first bit set to 1, only fonts whose x- and y-aspect exactly match the device are selected.

Returns Previous font-mapper flag.

SetMenu

Declaration `function SetMenu(Wnd: HWnd; Menu: HMenu): Bool;`

Sets and redraws a window's menu to the menu specified by *Menu*. The previous menu is not destroyed.

Parameters *Wnd*: Window identifier.

Menu: New menu or 0 to remove current menu.

Returns Non-zero if successful; zero if not.

See also *DestroyMenu*.

SetMenuItemBitmaps

Declaration `function SetMenuItemBitmaps(Menu: HMenu; Position, Flags: Word; BitmapUnchecked, BitmapChecked: HBitmap): Bool;`

Associates two bitmaps with a menu item, one is displayed when the item is checked and another when the item is unchecked.

Parameters *Menu*: Menu identifier.

Position: Command ID or position of menu item.

Flags: One of *mf_ByCommand* or *mf_ByPosition*. See "(MF_) Menu flags" in Chapter 1.

BitmapUnchecked: *HBitmap* to be displayed when item not checked or 0 to display nothing.

BitmapChecked: *HBitmap* to be displayed when item checked or 0 to display nothing. If *BitmapUnchecked* and *BitmapChecked* are 0, Windows uses default checkmark.

Returns Non-zero if successful; zero if not.

SetMessageQueue

Declaration `function SetMessageQueue(Msg: Integer): Bool;`

Creates a new application message queue of the specified size. The old queue is deleted.

Parameters *Msg*: Size of queue.

Returns Non-zero if successful; zero if not.

SetMetaFileBits

Declaration `function SetMetaFileBits(Mem: THandle): THandle;`

Creates a memory metafile from the data specified by *Mem*.

Parameters *Mem*: Global memory block containing metafile data previously created by *GetMetaFileBits*.

Returns Memory metafile identifier if successful; 0 if not.

SetPaletteEntries

Declaration `function SetPaletteEntries(Palette: HPalette; StartIndex, NumEntries: Word; var PaletteEntries): Word;`

Sets logical palette entries in the specified range, to the values in *PaletteEntries*.

Parameters *Palette*: Logical palette identifier.

StartIndex: First entry to set.

NumEntries: Number of entries to set.

PaletteEntries: Array of *TPaletteEntry* structure.

Returns Actual number of entries set; zero if error.

SetParent

Declaration `function SetParent(WndChild, WndNewParent: HWnd): HWnd;`

Changes the parent of a child window to *WndNewParent*.

Parameters *WndChild*: Child window identifier.

WndNewParent: Parent window identifier.

Returns Previous parent window.

SetPixel

Declaration `function SetPixel(DC: HDC; X, Y: Integer; Color: TColorRef): Longint;`

Paints the pixel at the specified point.

Parameters *DC*: Device context identifier.

X, Y: Logical coordinates of point.

Color: *TColorRef* specifying the color to paint the point.

Returns Actual *TColorRef* used to paint; –1 if point outside clipping region.

SetPolyFillMode

Declaration `function SetPolyFillMode(DC: HDC; PolyFillMode: Integer): Integer;`

Sets the polygon filling mode used by GDI functions which use the polygon algorithm for computing interior points.

Parameters *DC*: Device context identifier.

PolyFillMode: One of *Alternate* or *Winding*. See "PolyFill modes" in Chapter 1.

Returns Previous filling mode if successful; zero if not.

SetProp

Declaration `function SetProp(Wnd: HWnd; Str: PChar; Data: THandle): Bool;`

Adds or changes an entry specified by *Str*, to a window's property list.

Parameters *Wnd*: Window identifier.

Str: String (null-terminated) or atom value created by calling *AddAtom*.

Data: Associated property data handle.

Returns Non-zero if added; zero if not.

SetRect

Declaration `procedure SetRect(var Rect: TRect; X1, Y1, X2, Y2: Integer);`

Fills *Rect* with the specified coordinates.

Parameters *Rect*: Receiving *TRect* structure.

X1, Y1: Upper-left corner of rectangle.

X2, Y2: Lower-left corner of rectangle.

SetRectEmpty

Declaration `procedure SetRectEmpty(var Rect: TRect);`

Sets all *Rect* coordinates to zero.

Parameters *Rect*: Receiving *TRect* structure.

SetRectRgn

Declaration `procedure SetRectRgn(Rgn: HRgn; X1, Y1, X2, Y2: Integer);`

Uses the space allocated for *Rgn* to create a rectangular region with the specified size.

Parameters *Rgn*: Region identifier.

X1, Y1: Upper-left corner of rectangle region.

X2, Y2: Lower-left corner of rectangle region.

See also *CreateRectRgn*.

SetResourceHandler

Declaration
```
function SetResourceHandler(Instance: THandle; ResType: Pointer;
LoadFunc: TFarProc): TFarProc;
```

Sets up a callback which is called by Windows when a resource is being locked (i.e. *LockResource*). The callback is passed a *Mem* to the stored resource, *Instance*, and *ResInfo* (from *FindResource*).

Parameters *Instance*: Module instance whose executable file contains the resource.

ResType: Pointer to short integer specifying a resource type.

LoadFunc: Callback function's procedure-instance address.

Returns Pointer to callback function.

SetROP2

Declaration `function SetROP2(DC: HDC; DrawMode: Integer): Integer;`

Sets the current drawing mode to the value specified by *DrawMode*. This mode specifies how object interiors and pens are combined with colors already on the display surface.

Parameters *DC*: Device context identifier.

DrawMode: Any one of the *r2_* constants. See "(r2_) Binary raster operations" in Chapter 1.

Returns Previous drawing mode.

SetScrollPos

Declaration `function SetScrollPos(Wnd: HWnd; Bar, Pos: Integer; Redraw: Bool): Integer;`

Sets a scroll bar's thumb to *Pos*.

Parameters *Wnd*: Window identifier or scroll bar control identifier if *Bar* set to *sb_Ctl*.

Bar: One of *sb_Ctl*, *sb_Horz*, or *sb_Vert*. See "(SB_) Scroll bar constants" in Chapter 1.

Pos: New position.

Redraw: Non-zero to redraw scroll bar.

Returns Previous position of scroll bar thumb.

SetScrollRange

Declaration `procedure SetScrollRange(Wnd: HWnd; Bar, MinPos, MaxPos: Integer; Redraw: Bool);`

Sets a scroll bar's minimum and maximum position values.

Parameters *Wnd*: Window identifier or scroll bar control identifier if *Bar* set to *sb_Ctl*.

Bar: One of *sb_Ctl*, *sb_Horz*, or *sb_Vert*. See "(sb_) Scroll bar constants" in Chapter 1.

MinPos: Minimum scrolling position.

MaxPos: Maximum scrolling position or zero and *MinPos* set to zero to hide scroll bar.

Redraw: Non-zero to redraw scroll bar.

SetSoundNoise

Declaration `function SetSoundNoise(Source, Duration: Integer): Integer;`

Sets a noise source and duration values for the play device.

Parameters *Source*: Any one of: *s_Period512*, *s_Period1024*, *s_Period2048*, *s_PeriodVoice*, *s_White512*, *s_White1024*, *s_White2048*, or *s_WhiteVoice*. See "(s_) Sound constants" in Chapter 1.

Duration: Noise duration (in clock ticks).

Returns Zero if successful; *s_SerDSR* if invalid *Source*.

SetStretchBltMode

Declaration `function SetStretchBltMode(DC: HDC; StretchMode: Integer): Integer;`

Sets the stretching mode used by *StretchMode* to contract a bitmap.

Parameters *DC*: Device context identifier.

StretchMode: One of *BlackOnWhite*, *ColorOnColor*, or *WhiteOnBlack*. See "StretchBlt modes" in Chapter 1.

Returns Previous stretching mode.

SetSwapAreaSize

Declaration `function SetSwapAreaSize(Size: Word): Longint;`

Increases the amount of memory, up to one-half of remaining space after Windows is loaded, available to an application for its code segments.

Parameters *Size*: Number of 16-byte paragraphs.

Returns Actual number of paragraphs obtained and maximum size available in low- and high-order word, respectively.

SetSysColors

Declaration `procedure SetSysColors(Changes: Integer; var SysColor: Integer; var ColorValues: Longint);`

Globally sets the system colors for the display elements specified in *SysColor*.

Parameters *Change*: Number of system colors to change.

SysColor: Integer array whose indexes are *color_* constants. See "(color_) System color codes" in Chapter 1.

ColorValues: *Longint* array containing corresponding RGB color value for each color index in *SysColor*.

SetSysModalWindow

Declaration `function SetSysModalWindow(Wnd: HWnd): HWnd;`

Makes a *Wnd* a system-modal window. The system-modal state is ended when the window is destroyed.

Parameters *Wnd*: Window identifier.

Returns Previous system-modal window.

SetSystemPaletteUse

Declaration `function SetSystemPaletteUse(DC: HDC; Usage: Word): Word;`

Allows full system palette usage by an application whose window currently has the input focus.

Parameters *DC*: Device context identifier.

Usage: One of *syspal_NoStatic* or *syspal_Static*. See "(*syspal_*) System palette flags" in Chapter 1.

Returns Previous system palette usage.

See also *GetSysColor, SetSysColors, UnrealizeObject, wm_SysColorChange.*

SetTextAlign

Declaration `function SetTextAlign(DC: HDC; Flags: Word): Word;`

Sets text-alignment flags used by *TextOut* and *ExtTextOut* for positioning text in relationship to its bounding rectangle.

Parameters *DC*: Device or display context.

Flags: A combination of these constants: *ta_BaseLine, ta_Bottom, ta_Center, ta_Left, ta_NoUpdateCP, ta_Right, ta_Top, ta_UpdateCP*. See "(ta_) Text alignment options" in Chapter 1.

Returns Previous horizontal and vertical alignment in low- and high-order byte, respectively.

SetTextCharacterExtra

Declaration `function SetTextCharacterExtra(DC: HDC; CharExtra: Integer): Integer;`

Sets the amount of space added to each character when GDI writes a line of text.

Parameters *DC*: Device context identifier.

CharExtra: Amount (in logical units) of inter-character spacing.

Returns Previous inter-character spacing.

SetTextColor

Declaration `function SetTextColor(DC: HDC; Color: TColorRef): Longint;`

Sets the text color or nearest device supported color used by *TextOut* and *ExtTextOut* to draw a character's face. Also used by GDI to convert bitmaps from color to monochrome and vice versa.

Parameters *DC*: Device context identifier.

Color: Text *TColorRef*.

Returns Previous text RGB color value.

See also *SetBkColor, SetBkMode*.

SetTextJustification

Declaration `function SetTextJustification(DC: HDC; BreakExtra, BreakCount: Integer): Integer;`

Defines justification parameters used by GDI to justify a line of text.

Parameters *DC*: Device context identifier.

BreakExtra: Extra line space (in logical units) to be added.

BreakCount: Number of break characters (usually space character) in line.

Returns 1 if successful; zero if not.

See also *GetTextExtent, GetTextMetrics, TextOut.*

SetTimer

Declaration `function SetTimer(Wnd: HWnd; IDEvent: Integer; Elapse: Word; TimerFunc: TFarProc): Word;`

Creates a system timer which causes a *wm_Timer* message to be sent to an application at the interval specified by *Elapse.*

Parameters *Wnd*: Window identifier or 0 for no associated window.

IDEvent: Non-zero timer-event identifier or ignored if *Wnd* is 0.

Elapse: Number of milliseconds between timer events.

TimerFunc: Callback function's procedure-instance address or **nil** to place *wm_Timer* messages in application queue.

Returns *IDEvent* if *Wnd* is not 0; new timer event if otherwise; zero if error.

See also *wm_Timer.*

SetViewportExt

Declaration `function SetViewportExt(DC: HDC; X, Y: Integer): Longint;`

Sets a viewport's x- and y-extents which defines how GDI stretches or compresses logical units to fit device units.

Parameters *DC*: Device context identifier.

X, Y: New viewport extents (in device units).

Returns Previous x- and y-extents in low- and high-order words, respectively.

SetViewportOrg

Declaration

```
function SetViewportOrg(DC: HDC; X, Y: Integer): Longint;
```

Sets a viewport's origin which defines how GDI maps logical coordinates to points in device coordinates.

Parameters *DC*: Device context identifier.

X, Y: New viewport origin (in device units).

Returns X- and y-coordinates of previous origin in low- and high-order words, respectively.

SetVoiceAccent

Declaration

```
function SetVoiceAccent(Voice, Tempo, Volume, Mode, Pitch: Integer):
Integer;
```

Replaces the envelope in a voice queue.

Parameters *Voice*: A voice queue (starting from 1).

Tempo: Quarter notes played per minute (default, 120).

Volume: Volume level (0 to 255).

Mode: One of the constants *s_Legato*, *s_Normal*, or *s_Staccato*. See "(s_) Sound constants" in Chapter 1.

Pitch: Pitch of played notes (0 to 83).

Returns Zero if successful; one of these negative constants if not: *s_SerDMD*, *s_SerDTP*, *s_SerDVL*, or *s_SerQFUL*. See "(s_) Sound constants" in Chapter 1.

SetVoiceEnvelope

Declaration `function SetVoiceEnvelope(Voice, Shape, RepeatCount: Integer): Integer;`

Places a voice envelope in the voice queue, replacing the existing one.

Parameters *Voice*: A voice queue.

Shape: OEM wave-shape table index.

RepeatCount: Number of repetitions of wave-shape per note.

Returns Zero if successful; one of these negative constants if not: *s_SerQFUL* or *s_SerDSH*. See "(s_) Sound constants" in Chapter 1.

SetVoiceNote

Declaration `function SetVoiceNote(Voice, Value, Length, Cdots: Integer): Integer;`

Places a note on a voice queue with the specified qualities.

Parameters *Voice*: A voice queue.

Value: A note (1 of 84) or zero for rest.

Length: Reciprocal of note duration.

Cdots: Note duration in dots: (*Length* * (*Cdots* * 3/2)).

Returns Zero if successful; one of these negative constants if not: *s_serDCC*, *s_serDLN*, *s_serBDNT*, or *s_serQFUL*. See "(s_) Sound constants" in Chapter 1.

SetVoiceQueueSize

Declaration `function SetVoiceQueueSize(Voice, Bytes: Integer): Integer;`

Sets the size of non-playing voice queue. The default is 192 bytes or approximately 32 notes.

Parameters *Voice*: A voice queue.

Bytes: Size (in bytes) of voice queue.

Returns Zero if successful; one of these negative constants if not: *s_serMACT* or *s_serOFM*. See "(s_) Sound constants" in Chapter 1.

SetVoiceSound

Declaration `function SetVoiceSound(Voice: Longint; Frequency: Longint; Duration: Integer): Integer;`

Places the specified sound frequency and duration on a voice queue.

Parameters *Voice*: A voice queue.

Frequency: Hertz and fractional frequency in high- and low-order words, respectively.

Duration: Sound duration (in clock ticks).

Returns Zero if successful; one of these negative constants if not: *s_serDDR*, *s_serDFQ*, *s_serDVL*, or *s_serQFUL*. See "(s_) Sound constants" in Chapter 1.

SetVoiceThreshold

Declaration `function SetVoiceThreshold(Voice, Notes: Integer): Integer;`

Sets the threshold level for a voice queue. If the number of notes in queue drop below the threshold the threshold flag is set.

Parameters *Voice*: A voice queue.

Notes: Number of notes specifying the threshold level.

Returns Zero if successful; 1 if Notes out of range.

See also *GetThresholdStatus*.

SetWindowExt

Declaration `function SetWindowExt(DC: HDC; X, Y: Integer): Longint;`

Sets a window's x- and y-extents. This, along with the viewport extents, defines how GDI stretches or compresses logical units to fit device units.

Parameters *DC*: Device context identifier.

X, Y: Window extents.

Returns Previous x- and y-extents in low- and high-order words, respectively; zero if error.

SetWindowLong

Declaration

```
function SetWindowLong(Wnd: HWnd; Index: Integer; NewLong: Longint):
Longint;
```

Replaces an attribute of a window's window-class structure with a new value.

Parameters *Wnd*: Window identifier.

Index: *gwl_ExStyle*, *gwl_Style*, *gwl_WndProc* or positive byte offset to access extra four-byte values. See "(gwl_) Window field offsets" in Chapter 1.

NewLong: Replacement value.

Returns Previous value.

SetWindowOrg

Declaration

```
function SetWindowOrg(DC: HDC; X, Y: Integer): Longint;
```

Sets a window's origin within the viewport of the specified device context.

Parameters *DC*: Device context identifier.

X, *Y*: New window origin.

Returns Previous x- and y-coordinate in low-and high-order words, respectively.

SetWindowPos

Declaration

```
procedure SetWindowPos(Wnd, WndInsertAfter: HWnd; X, Y, cx, cy: Integer;
Flags: Word);
```

Changes the size, position, and ordering of a window.

Parameters *Wnd*: Window identifier.

WndInsertAfter: Preceding window in window-manager's list.

X, *Y*: Upper left corner.

cx: New window width.

cy: New window height.

Flags: One of: *swp_DrawFrame, swp_HideWindow, swp_NoActivate, swp_NoRemove, swp_NoSize, swp_NoRedraw, swp_NoZOrder, swp_ShowWindow*. See "(swp_) Set window position flags" in Chapter 1.

SetWindowsHook

Declaration `function SetWindowsHook(FilterType: Integer; FilterFunc: TFarProc): TFarProc;`

Installs a filter function in the chain of filter functions specified by *FilterType*. The filter function is passed an *Code, wParam,* and *lParam* whose values are filter type dependent.

Parameters *FilterType*: One of: *wh_CallWndProc, wh_GetMessage, wh_JournalPlayback, wh_JournalRecord, wh_Keyboard, wh_MsgFilter,* or *wh_SysMsgFilter*. See "(wh_) Windows hook codes" in Chapter 1.

FilterFunc: Filter function's procedure-instance address.

Returns Procedure-instance address of previously installed filter function; **nil** no previous filter.

See also *DefHookProc*.

SetWindowText

Declaration `procedure SetWindowText(Wnd: HWnd; Str: PChar);`

Sets the caption title for a window or text of a control with the string specified by *Str*.

Parameters *Wnd*: Window or control identifier.

Str: String (null-terminated).

SetWindowWord

Declaration `function SetWindowWord(Wnd: HWnd; Index: Integer; NewWord: Word): Word;`

Changes the value of an attribute specified by *Index*, in a window's window-class structure.

Parameters *Wnd*: Window identifier.

Index: *gww_HInstance*, *gww_HWndParent*, *gww_ID*, or positive byte offset to access extra two-byte values. See "(gww_) Window field offsets" in Chapter 1.

NewWord: Replacement value.

Returns Previous value.

ShowCaret

Declaration `procedure ShowCaret(Wnd: HWnd);`

Shows a caret which is owned by *Wnd* on the display.

Parameters *Wnd*: Window identifier or 0 for a window in the current task.

ShowCursor

Declaration `function ShowCursor(Show: Bool): Integer;`

Shows the cursor if its display count (initially set to zero) is greater or equal to zero. Otherwise the cursor is hidden.

Parameters *Show*: Non-zero to increment display count or zero to decrement display count.

Returns New display count.

ShowOwnedPopups

Declaration `procedure ShowOwnedPopups(Wnd: HWnd; Show: Bool);`

Shows or hides, as specified by *Show*, all popup windows associated with the given window.

Parameters *Wnd*: Window identifier.

Show: Non-zero to show all hidden popups or zero to hide all visible popups.

ShowScrollBar

Declaration `procedure ShowScrollBar(Wnd: HWnd; Bar: Word; Show: Bool);`

Shows or hides a scroll bar as specified by *Show*.

Parameters *Wnd*: Window identifier or scroll bar control if *Bar* set to *sb_Ctl*.

Bar: One of *sb_Both*, *sb_Ctl*, *sb_Horz*, or *sb_Vert*. See "(SB_) Scroll bar constants" in Chapter 1.

Show: Non-zero to show scroll bar or zero to hide scroll bar.

ShowWindow

Declaration `function ShowWindow(Wnd: HWnd; CmdShow: Integer): Bool;`

Shows or hides a window in the manner specified by *CmdShow*.

Parameters *Wnd*: Window identifier.

CmdShow: One of the *sw_* constants. See "(SW_) Show window constants" in Chapter 1.

Returns Non-zero if window was previously visible; zero if previously hidden.

SizeofResource

Declaration `function SizeofResource(Instance, ResInfo: THandle): Word;`

Retrieves the size of a resource. The returned size may larger due to alignment factors.

Parameters *Instance*: Module instance whose executable file contains the resource.

ResInfo: Desired resource, returned by *FindResource*.

Returns Size (in bytes) of resource; zero if not found.

See also *AccessResource*.

StartSound

Declaration `function StartSound: Integer;`

Non-destructively plays all voice queue.

Returns Not used.

StopSound

Declaration `function StopSound: Integer;`

Stops playing all voice queues, flushes queue contents, and turns off voice sound drivers.

Returns Not used.

StretchBlt

Declaration `function StretchBlt(DestDC: HDC; X, Y, Width, Height: Integer; SrcDC: HDC; XSrc, YSrc, SrcWidth, SrcHeight: Integer; Rop: Longint): Bool;`

Moves a bitmap, stretching or compressing as required, from a source rectangle to a destination rectangle. The source and destination are combined as specified by *Rop*.

Parameters *DestDC*: Receiving device context.

X, Y: Upper-left corner of destination rectangle.

Width: Destination rectangle width (in logical units).

Height: Destination rectangle height (in logical units).

SrcDC: Source bitmap device context.

XSrc, YSrc: Upper-left corner of source rectangle.

SrcWidth: Source rectangle width (in logical units).

SrcHeight: Source rectangle height (in logical units).

Rop: Raster operation to be performed. See "Ternary raster operations" in Chapter 1.

Returns Non-zero if bitmap is drawn; zero if not.

See also *SetStretchBltMode*.

StretchDIBits

Declaration

```
function StretchDIBits(DC: HDC; DestX, DestY, DestWidth, DestHeight,
SrcX, SrcY, SrcWidth, SrcHeight: Word; Bits: Pointer; var BitsInfo:
TBitmapInfo; Usage: Word; Rop: DWord): Integer;
```

Moves a device-independent bitmap (DIB), stretching or compressing as required, from a source rectangle to a destination rectangle. The source and destination are combined as specified by *Rop*.

Parameters *DC*: Receiving device context.

DestX, DestY: Destination rectangle origin (in logical units).

DestWidth: Destination rectangle x-extent (in logical units).

DestHeight: Destination rectangle y-extent (in logical units).

SrcX, SrcY: Source rectangle origin (in pixels) in the DIB.

SrcWidth: Source rectangle width (in pixels).

SrcHeight: Source rectangle height (in pixels).

Bits: Byte array containing DIB bits.

Usage: *DIB_RGB_Colors* specifies *BitsInfo.bmiColors* field contains RGB values or *DIB_Pal_Colors* specifies indexes to currently realized logical palette. See "(DIB_) Color table identifiers" in Chapter 1, "Windows styles and constants."

Rop: One of the ternary raster operation constants. See "Ternary raster operations" in Chapter 1, "Windows styles and constants."

Returns Number of scan lines copied.

SwapMouseButton

Declaration

```
function SwapMouseButton(Swap: Bool): Bool;
```

Reverses or restores the meaning of the left and right mouse button as specified by *Swap*.

Parameters *Swap*: Non-zero to swap button meanings or zero to restore original meanings.

Returns Non-zero if meanings reversed; zero if not.

SwapRecording

Declaration

```
procedure SwapRecording(Flag: Word);
```

Used to begin or end analyzing swapping behavior when running Windows *Swap* program.

Parameters *Flag*: (0) stop analyzing, (1) record swap calls and discard swap returns, (2) same as 1 and calls through thunks.

SwitchStackBack

Declaration

```
procedure SwitchStackBack;
```

Restores the current task's stack to its data segment preserving the contents of AX:DX registers.

See also *SwitchStackTo*

SwitchStackTo

Declaration `procedure SwitchStackTo(StackSegment, StackPointer, StackTop: Word);`

Changes the stack of the current task to *StackSegment*. Can be used to set the stack of a DLL to its data segment if it has functions that assume DS=SS.

Parameters *StackSegment*: Data segment to contain the stack.

StackPointer: Beginning stack offset into *StackSegment*.

StackTop: Top of stack offset from *StackPointer*.

See also *SwitchStackBack*

SyncAllVoices

Declaration `function SyncAllVoices: Integer;`

Places a sync mark in all voice queues.

Returns Zero if successful; *s_SerQFUL* if voice queue full. See "(s_) Sound constants" in Chapter 1.

S

TabbedTextOut

Declaration
```
function TabbedTextOut(DC: HDC; X, Y: Integer; Str: PChar; Count,
TabPositions: Integer; var TabStopPositions; TabOrigin: Integer):
Longint;
```

Draws a line of text with tabs expanded as specified in *TabStopPositions*, using the selected font.

Parameters *DC*: Device context identifier.

X, Y: String starting point.

Str: String to be drawn.

Count: Size (in characters) of *Str*.

TabPositions: Number of tab-stop positions in *TabStopPositions* or zero if tab-stops are expanded eight average character widths.

TabStopPositions: Integer array containing ascending tab-stop positions (in pixels).

TabOrigin: Starting position (in logical units) from which tabs are expanded.

Returns Not used.

TextOut

Declaration

```
function TextOut(DC: HDC; X, Y: Integer; Str: PChar; Count: Integer):
Bool;
```

Draws a line of text using the currently selected font.

Parameters *DC*: Device context identifier.

X, Y: String starting point.

Str: String to be drawn.

Count: Size (in characters) of *Str*.

Returns Non-zero if drawn; zero if not.

See also *SetTextAlign*.

Throw

Declaration

```
procedure Throw(var CatchBuf: TCatchBuf; ThrowBack: Integer);
```

Restores an application's execution environment. Execution continues at the *Catch* function that originally saved the environment to *CatchBuf*.

Parameters *CatchBuf*: *TCatchBuf* containing the execution environment.

ThrowBack: Value returned to *Catch* function.

ToAscii

Declaration

```
function ToAscii(VirtKey, ScanCode: Word; KeyState: PChar; CharBuff:
Pointer; Flags: Word): Integer;
```

Translates *VirtKey* and current keyboard state into corresponding ANSI characters.

Parameters

VirtKey: Virtual-key code.

ScanCode: Raw key scan code, high bit set if key up.

KeyState: Array of 256 Bytes containing the state of each key, high bit set if key up.

CharBuff: Pointer to a 32-bit receiving buffer.

Flags: Not used.

Returns (2) accent and dead key copied to *CharBuff*, (1) one ANSI character copied to *CharBuff*, (0) translation not possible with current keyboard state.

TrackPopupMenu

Declaration

```
function TrackPopupMenu(Menu: HMenu; Flags: Word; x, y, cx: Integer; Wnd:
HWnd; var Rect: TRect): Bool;
```

Displays a floating popup menu and tracks item selection. Floating popup menus may appear anywhere on the screen.

Parameters

Menu: Popup menu identifier.

Flags: Set to zero, not used.

x, y: Upper-left position (in screen coordinates) of menu.

cx: Menu width (in screen units) or zero for default.

Wnd: Owning window of the popup menu, to receive *wm_Command* messages.

Rect: *TRect* defining mouse area where menu remains visible if the user releases the mouse button.

Returns Non-zero if successful; zero if not.

See also *CreatePopupMenu, GetSubMenu.*

TranslateAccelerator

Declaration `function TranslateAccelerator(Wnd: HWnd; AccTable: THandle; var Msg: TMsg): Integer;`

Translates keyboard accelerators (*wm_KeyUp* and *wm_KeyDown*) into menu command messages, *wm_Command* and *wm_SysCommand*, which are then sent directly to the window.

Parameters *Wnd*: Window identifier.

AccTable: Accelerator table identifier (returned by *LoadAccelerators*).

Msg: *TMsg* information retrieved from *GetMessage* or *PeekMessage*.

Returns Non-zero if translation occurred; zero if not.

TranslateMDISysAccel

Declaration `function TranslateMDISysAccel(Wnd: HWnd; var Msg: TMsg): Bool;`

Translates keyboard accelerators for MDI child window System-menu *wm_SysCommand* messages which are then sent directly to the window.

Parameters *Wnd*: MDI client parent window.

Msg: *TMsg* information retrieved from *GetMessage* or *PeekMessage*.

Returns Non-zero if translation occurred; zero if not.

TranslateMessage

Declaration `function TranslateMessage(var Msg: TMsg): Bool;`

Translates *wm_KeyDown/Up* combinations to *wm_Char* or *wm_DeadChar* and *wm_SysKeyDown/Up* combinations to *wm_SysChar* or *wm_SysDeadChar* and posts the character message to the application queue.

Parameters *Msg*: *TMsg* information retrieved from *GetMessage* or *PeekMessage*.

Returns Non-zero if translation occurred; zero if not.

TransmitCommChar

Declaration `function TransmitCommChar(Cid: Integer; AChar: Char): Integer;`

Places *AChar* at the head of the communications device's transmit queue for immediate transmission.

Parameters *Cid*: Communications device.

AChar: Character to transmit.

Returns Zero if successful; negative if cannot transmit because previous character has not been sent yet.

See also *OpenComm*.

UngetCommChar

Declaration `function UngetCommChar(Cid: Integer; AChar: Char): Integer;`

Places *AChar* back into communications device's receiving queue.

Parameters *Cid*: Communications device.

AChar: Character to unget.

Returns Zero if successful; negative if not.

See also *OpenComm*.

UnhookWindowsHook

Declaration `function UnhookWindowsHook(Hook: Integer; HookFunc: TFarProc): Bool;`

Removes a hook function from the chain of hook functions specified by *Hook*.

Parameters *Hook*: One of: *wh_CallWndProc, wh_GetMessage, wh_JournalPlayback, wh_JournalRecord, wh_Keyboard,* or *wh_MsgFilter*. See "(wh_) Windows hook codes" in Chapter 1.

HookFunc: Hook function's procedure-instance address.

Returns Non-zero if successful; zero if not.

UnionRect

Declaration `function UnionRect(var DestRect, Src1Rect, Src2Rect: LPRect): Integer;`

Creates a union of two rectangles and stores the result in *DestRect*.

Parameters *DestRect*: Result *TRect* structure.

Src1Rect: Source *TRect* structure 1.

Src2Rect: Source *TRect* structure 2.

Returns Non-zero if union not empty; zero if empty.

UnlockData

Declaration `function UnlockData(Dummy: Integer): THandle;`

Unlocks the current, moveable data segment.

Parameters *Dummy*: Not used. Set to zero.

Returns Identifier for the unlocked segment, or zero if unsuccessful.

UnlockResource

Declaration `function UnlockResource(RezData: THandle): Bool;`

Unlocks the *RezData* resource and decrements its reference counter.

Parameters *RezData*: Global memory block Identifier.

Returns Zero if the reference counter is zero; non-zero if not.

UnlockSegment

Declaration `function UnlockSegment(Segment: Word): THandle;`

Unlocks a segment specified by *Segment*.

Parameters *Segment*: Segment address or –1 to unlock current data segment.

Returns Zero if reference count decremented to zero; Non-zero if not.

See also *LockSegment*.

UnrealizeObject

Declaration `function UnrealizeObject(hObject: HBrush): Bool;`

Directs GDI to reset the origin the next time selected if *hObject* is a brush or realize the palette if *hObject* is a logical palette.

Parameters *hObject*: Object to be reset.

Returns Non-zero if successful; zero if not.

UnregisterClass

Declaration `function UnregisterClass(ClassName: PChar; Instance: THandle): Bool;`

Deletes a window class from the window-class table and frees all associated memory.

Parameters *ClassName*: Class name (null-terminated) of a previously registered class.

Instance: Module instance that created the class.

Returns Non-zero if successful; zero if invalid *ClassName* or a window of the class exits.

See also *RegisterClass*.

UpdateColors

Declaration `function UpdateColors(DC: HDC): Integer;`

Updates the client area by matching the current colors of the client area pixel-by-pixel with the system palette.

Parameters *DC*: Device context identifier.

Returns Not used.

UpdateWindow

Declaration `procedure UpdateWindow(Wnd: HWnd);`

Sends a *wm_Paint* message directly the given window's window function if the update region of the window is not empty.

Parameters *Wnd*: Window identifier.

ValidateCodeSegments

Declaration `procedure ValidateCodeSegments;`

Outputs debugging information to a terminal if any code segments have been altered by random memory overwrites. Only available in the debugging version of Windows. To disable the function, set the *EnableSegmentChecksum* flag in the [kernel] section of WIN.INI to 0.

Not used in Standard and 386-Enhanced modes of Windows.

ValidateFreeSpaces

Declaration `function ValidateFreeSpaces: Pointer;`

Checks all free memory segments for valid contents. This function works only in the debugging version of Windows.

Returns Not used.

ValidateRect

Declaration `procedure ValidateRect(Wnd: HWnd; Rect: LPRect);`

Validates the client area by removing *Rect* from the window's update region.

Parameters *Wnd*: Window identifier.

Rect: *TRect* (in client coordinates) to be removed from update region or **nil** for entire client area.

See also *BeginPaint*.

ValidateRgn

Declaration `procedure ValidateRgn(Wnd: HWnd; Rgn: HRgn);`

Validates the client area by removing the region specified by *Rgn* from the window's update region.

Parameters *Wnd*: Window identifier.

Rgn: Region identifier (in client coordinates).

VkKeyScan

Declaration `function VkKeyScan(AChar: Word): Word;`

Translates *AChar* into its corresponding virtual-key code and shift state.

Parameters *AChar*: ANSI character to find corresponding virtual-key code.

Returns Virtual-key code in the low-order byte; the following shift states in the high-order byte: (0) no shift, (1) shifted, (2) control char, (6) *Ctrl+Alt*, (7) *Shift+Ctrl+Alt*, or (3), (4), or (5), which are not used for characters. In the case of an error, both bytes hold –1.

WaitMessage

Declaration `procedure WaitMessage;`

Yields control to other applications and does not return until a message becomes available in the application queue.

WaitSoundState

Declaration `function WaitSoundState(State: Integer): Integer;`

Waits for the play driver to enter the state specified by *State*.

Parameters *State*: One of *s_AllThreshold*, *s_QueueEmpty*, or *s_Threshold*. See "(s_) Sound constants" in Chapter 1.

Returns Zero if successful; *s_SerDST* if invalid *State*.

WindowFromPoint

Declaration `function WindowFromPoint(Point: TPoint): HWnd;`

Determines the window that contains the specified point.

Parameters *Point*: *TPoint* to be checked (in screen coordinates).

Returns Window identifier; 0 if no window at specified point.

WinExec

Declaration `function WinExec(CmdLine: PChar; CmdShow: Word): Word;`

Executes a application specified by *CmdLine*.

Parameters *CmdLine*: Command line to execute the application (null-terminated).

CmdShow: Specifies how application window is to be initially shown (see *ShowWindow*).

Returns Value greater than 32 if successful; if not returns one of: (0) no memory, (5) attempted to dynamically link to a task, (6) library has multiple data segments, (10) Windows version incorrect, (11) invalid EXE file, (12) OS/2 application, (13) DOS 4.0 application, (14) unknown EXE type, or (15) not a protect-mode application.

WinHelp

Declaration `function WinHelp(Wnd: HWnd; HelpFile: PChar; Command: Word; Data: Longint): Bool;`

Invokes Windows help engine with a command specified by *Command*.

Parameters *Wnd*: Window identifier.

HelpFile: Name (null-terminated) of help file including path, if needed.

Command: One of *help_Context*, *help_HelpOnHelp*, *help_Index*, *help_Key*, *help_Quit*, or *help_SetIndex*. See "(help_) Help commands" in Chapter 1.

Data: Context identifier number if *Command* set to *help_Context* or help topic keyword (null-terminated) if *Command* set to *help_Key*.

Returns Non-zero if successful; zero if not.

WriteComm

Declaration `function WriteComm(Cid: Integer; Buf: PChar; Size: Integer): Integer;`

Outputs the buffer specified by *Buf* to a communications device.

Parameters *Cid*: Communications device.

Buf: Buffer containing characters to be written.

Size: Number of characters to output.

Returns Actual number of characters written; negative value if error whose absolute value is the number characters written before the error occurred.

WritePrivateProfileString

Declaration `function WritePrivateProfileString(ApplicationName, KeyName, Str, FileName: PChar): Bool;`

Searches *FileName* for the specified application heading and key name replacing the value with *Str*.

Parameters *ApplicationName*: Application heading name.

KeyName: Key name appearing under the application heading name or **nil** to delete entire section.

Str: New key value string or **nil** to delete key name.

FileName: Initialization file name (null-terminated).

Returns Non-zero if successful; zero if not.

WriteProfileString

Declaration `function WriteProfileString(ApplicationName, KeyName, Str: PChar): Bool;`

Searches WIN.INI for the specified application heading and key name replacing the value with *Str*.

Parameters *ApplicationName*: Application heading.

KeyName: Key name appearing under the application heading name or **nil** to delete entire application section.

Str: New key name value or **nil** to delete key name.

Returns Non-zero if successful; zero if not.

wvsprintf

Declaration `function wvsprintf(Output, Format, ArgList: PChar): Integer;`

Formats and stores a series of characters in a buffer.

Parameters *Output*: Buffer to receive formatted characters.

Format: Format control string.

ArgList: An array of arguments to the format control string.

Returns If successful, the number of characters in *Output*, not counting 0; less than the length of *Format* if not.

Yield

Declaration `function Yield: Bool;`

Halts the current task and starts a waiting task.

Returns Not used.

C H A P T E R

3

Windows message reference

Each alphabetical entry describes what the message does, values for each message field, and the return value expected or given by Windows. Any additional comments are included at the end of each description. *wParam* and *lParam* and the parameters each Windows message carries. For full details on the *TMsg* record, including the *wParam* and *lParam* fields, see "TMsg type" in Chapter 4, "Windows type reference."

bm_GetCheck

Determines whether a radio button or check box is checked.

Parameters *wParam*: Not used.

lParam: Not used.

Return value If the radio button or check box is checked, the return value is non-zero. Otherwise the return value is zero. The return value is always zero for a push button.

bm_GetState

Determines the state of a button control when a mouse button or *Spacebar* is pressed.

Parameters *wParam*: Not used.

lParam: Not used.

Return value The return value is non-zero if the button is a highlighted push button, the button has the input focus and a mouse button or *Spacebar* is pressed, or a mouse button is pressed when the cursor is over the button. Otherwise the return value is zero.

bm_SetCheck

Checks or removes a checkmark from a radio button or check box.

Parameters *wParam*: For two-state buttons and check boxes if *wParam* is zero, the checkmark (if any) is removed, otherwise a checkmark is added. For three-state buttons if *wParam* is zero, the checkmark (if any) and graying (if any) are removed. If *wParam* is 1, a checkmark is added. If *wParam* is 2, the button is grayed.

lParam: Not used.

Return value Not used.

bm_SetState

Changes the state of a button or check box.

Parameters *wParam*: If *wParam* is zero, the button or check box is drawn normally. If *wParam* is non-zero, the button is highlighted.

lParam: Not used.

Return value Not used.

bm_SetStyle

Changes the style of a button.

Parameters *wParam*: *wParam* is the new button style. See "(bs_) Button styles" in Chapter 1, "Windows styles and constants".

lParam: If *lParam* is zero, the button will not be immediately redrawn. If *lParam* is non-zero and the new button style is different than the current button style, the button will be redrawn.

Return value Not used.

cb_AddString

Adds a string to a combo box's list box.

Parameters *wParam*: Not used.

lParam: *lParam* is a pointer to a null-terminated string to be added.

Return value If successful, the index at which the string was added is returned, otherwise *cb_ErrSpace* is returned if there wasn't enough memory available to store the string or *cb_Err* is returned if there was an error.

Comments If the combo box's list box is not sorted, the string is put at the end of the list. If the combo box has the *cbs_OwnerDrawFixed* or *cbs_OwnerDrawVariable* style and does not have the *cbs_HasStrings* style, *lParam* is a 32-bit value which is stored instead of a string and each entry added is compared to other entries one or more times via a *wm_CompareItem* message sent to the owner of the combo box.

cb_DeleteString

Deletes a string from a combo box's list box.

Parameters *wParam*: *wParam* is the index of the list box entry to be deleted.

lParam: Not used.

Return value If *wParam* is a valid index, the number of entries remaining in the list is returned, otherwise *cb_Err* is returned.

Comments If the combo box has the *cbs_OwnerDrawFixed* or *cbs_OwnerDrawVariable* style and does not have the *cbs_HasStrings* style, the associated 32-bit value is deleted and a *wm_DeleteItem* message is sent to the owner of the combo box.

cb_Dir

Adds to a combo box's list box each file name in the current directory that matches a file specification and DOS file attribute.

Parameters *wParam*: *wParam* is the DOS file attribute.

lParam: *lParam* is a pointer to a null-terminated file specification string.

Return value If successful, the index of the last entry in the resulting list is returned, otherwise *cb_ErrSpace* is returned if there wasn't enough memory available to store the entries or *cb_Err* is returned if there was an error.

cb_FindString

Finds the first entry of a combo box's list box that matches a prefix string.

Parameters *wParam*: *wParam* is the index at which to start the search. The first entry looked at is the one after *wParam*. If the end of the list is reached, the search will continue with entry zero until the search index reaches *wParam*. If *wParam* is –1, the entire list is searched starting at entry zero.

lParam: *lParam* is a pointer to a null-terminated prefix string.

Return value If successful, the index of the first matching entry is returned, otherwise *cb_Err* is returned.

Comments If the combo box has the *cbs_OwnerDrawFixed* or *cbs_OwnerDrawVariable* style and does not have the *cbs_HasStrings* style, *lParam* is a 32-bit value which is compared to each associated 32-bit value in the list.

cb_GetCount

Returns the number of entries in a combo box's list box.

Parameters *wParam*: Not used.

lParam: Not used.

Return value The count of the list box entries is returned.

cb_GetCurSel

Returns the index of the currently selected entry in a combo box's list box.

Parameters *wParam*: Not used.

lParam: Not used.

Return value If no entry is selected, *cb_Err* is returned, otherwise the index of the currently selected entry is returned.

cb_GetEditSel

Returns the start and end indexes of the selected text in a combo box's edit control.

Parameters *wParam*: Not used.

lParam: Not used.

Return value If the combo box has no edit control, *cb_Err* is returned, otherwise the low-order word of the return value is the starting index and the high-order word of the return value is the ending index.

cb_GetItemData

Returns the 32-bit value associated with an entry in a combo box's list box.

Parameters *wParam*: *wParam* is the entry index.

lParam: Not used.

Return value If successful, the 32-bit associated value is returned, otherwise *cb_Err* is returned.

cb_GetLBText

Copies an entry from a combo box's list box into a supplied buffer.

Parameters *wParam*: *wParam* is the entry index.

lParam: *lParam* is a pointer to a buffer. The buffer must have enough space to contain the entry and a terminating null character.

Return value Not used.

Comments If the combo box has the *cbs_OwnerDrawFixed* or *cbs_OwnerDrawVariable* style and does not have the *cbs_HasStrings* style, the 32-bit value associated with the list entry is copied into the buffer.

cb_GetLBTextLen

Returns the length in bytes of an entry in a combo box's list box.

Parameters *wParam*: *wParam* is the entry index.

lParam: Not used.

Return value If *wParam* is a valid index, the length of the entry at that index is returned, otherwise *cb_Err* is returned.

cb_InsertString

Inserts a string into a combo box's list box without sorting.

Parameters *wParam*: If *wParam* is –1, the string is added to the end of the list, otherwise *wParam* is used as the index at which to insert the string.

lParam: *lParam* points to a null terminated string to be inserted.

Return value If successful, the index at which the string was inserted is returned, otherwise *cb_ErrSpace* is returned if there wasn't enough memory available to store the string or *cb_Err* is returned if there was an error.

cb_LimitText

Sets a limit to the number of characters which may be typed into a combo box's edit control.

Parameters *wParam*: *wParam* is the new maximum number of characters to be allowed. If *wParam* is zero there is no limit.

lParam: Not used.

Return value If successful, a non-zero value is returned, otherwise zero is returned. If the combo box has no edit control, *cb_Err* is returned.

cb_ResetContent

Removes all entries from a combo box's list box.

Parameters *wParam*: Not used.

lParam: Not used.

Return value Not used.

Comments If the combo box has the *cbs_OwnerDrawFixed* or *cbs_OwnerDrawVariable* style and does not have the *cbs_HasStrings* style, a *wm_DeleteItem* message is sent to the owner of the combo box for each entry.

cb_SelectString

Selects the first entry of a combo box's list box that matches a prefix string and updates the combo box's edit control or static-text control to reflect the selection.

Parameters *wParam*: *wParam* is the index at which to start the search. The first entry looked at is the one after *wParam*. If the end of the list is reached the search will continue with entry zero until the search index reaches *wParam*. If *wParam* is –1, the entire list is searched starting at entry zero.

lParam: *lParam* is a null-terminated prefix string.

Return value If successful, the index of the first matching entry is returned, otherwise *cb_Err* is returned and the current selection remains unchanged.

Comments If the combo box has the *cbs_OwnerDrawFixed* or *cbs_OwnerDrawVariable* style and does not have the *cbs_HasStrings* style, *lParam* is a 32-bit value which is compared to each associated 32-bit value in the list.

cb_SetCurSel

Selects an entry of a combo box's list box and updates the combo box's edit control or static-text control to reflect the selection.

Parameters *wParam*: *wParam* is the entry index. If *wParam* is –1, there is to be no selection.

lParam: Not used.

Return value if *wParam* is –1 or not a valid index, *cb_Err* is returned, otherwise the index of the selected entry is returned.

cb_SetEditSel

Sets the selected text in a combo box's edit control.

Parameters *wParam*: Not used.

lParamLo: *lParamLo* is the starting character index.

lParamHi: *lParamHi* is the ending character index.

Return value If successful, a non-zero value is returned, otherwise zero is returned. If the combo box has no edit control, *cb_Err* is returned.

cb_SetItemData

Sets the 32-bit value associated with an entry in a combo box's list box.

Parameters *wParam*: *wParam* is the entry index.

lParam: *lParam* is the new 32-bit value to be associated with the entry.

Return value *cb_Err* is returned if there is an error.

cb_ShowDropDown

Shows or hides a combo box's drop-down list box.

Parameters *wParam*: If *wParam* is zero, the drop-down list box is hidden, otherwise it is shown.

lParam: Not used.

Return value Not used.

Comments This message only applies to combo boxes created with the *cbs_DropDown* or *cbs_DropDownList* styles.

dm_GetDefID

Returns a dialog's default push-button control ID.

Parameters *wParam*: Not used.

lParam: Not used.

Return value If there is no default push-button ID for the dialog, the high-order word of the return value is zero, otherwise the high order word of the return value is *dc_HasDefID* and the low-order word of the return value is the default push-button ID.

dm_SetDefID

Sets a dialog's default push-button control ID.

Parameters *wParam*: *wParam* is the new default push-button ID.

lParam: Not used.

Return value Not used.

em_CanUndo

Determines whether an edit control can respond to an *em_Undo* message.

Parameters *wParam*: Not used.

lParam: Not used.

Return value If the edit control can respond to an *em_Undo* message, the return value is non-zero, otherwise the return value is zero.

em_EmptyUndoBuffer

Empties an edit control's undo buffer which disables its ability to undo the last edit.

Parameters *wParam*: Not used.

lParam: Not used.

Return value Not used.

Comments A *wm_SetText* or *em_SetHandle* message sent to an edit control causes the edit control's undo buffer to be emptied automatically.

em_FmtLines

Tells an edit control whether or not to add a special end-of-line character sequence to word wrapped text lines.

Parameters *wParam*: If *wParam* is non-zero, word wrapped lines are terminated with a carriage return, carriage return, line feed character sequence, otherwise

any carriage return, carriage return, line feed character sequence is removed from the text.

lParam: Not used.

Return value If the text was altered, a non-zero value is returned, otherwise zero is returned.

Comments Normal end-of-line character sequences (a single carriage return followed by a line feed) are not affected by this message. The size of the text is changed when the return value is non-zero. This message only applies to multiline edit controls.

em_GetHandle

Returns an edit control's buffer's handle. The buffer contains the edit control text.

Parameters *wParam*: Not used.

lParam: Not used.

Return value The edit control's buffer's handle is returned.

Comments This message may only be sent to an edit control which was created with the *ds_LocalEdit* style.

em_GetLine

Returns one line from an edit control.

Parameters *wParam*: *wParam* is the line number. Lines start at zero in an edit control.

lParam: *lParam* points to a buffer which is to contain the line. The first word of the buffer is the number of bytes to be transferred to the buffer.

Return value The actual number of bytes transferred is returned. There is no terminating null character added to the end of the buffer. This message only applies to multiline edit controls.

em_GetLineCount

Returns the number of text lines in the edit control.

Parameters *wParam*: Not used.

lParam: Not used.

Return value The number of text lines is returned.

Comments This message only applies to multiline edit controls.

em_GetModify

Returns an edit control's modify flag. The modify flag is set when the edit control's text is modified by entering new text or modifying the existing text or when an *em_SetModify* message is sent to the edit control.

Parameters *wParam*: Not used.

lParam: Not used.

Return value The edit control's modify flag is returned. Non-zero means the edit control's text has been modified, zero means it has not.

em_GetRect

Retrieves the formatting rectangle of an edit control.

Parameters *wParam*: Not used.

lParam: *lParam* points to a *TRect* data structure which is filled in by this message.

Return value Not used.

em_GetSel

Returns the start and end indexes of the selected text in an edit control.

Parameters *wParam*: Not used.

lParam: Not used.

Return value The low-order word of the return value is the starting index and the high-order word of the return value is the ending index.

em_LimitText

Sets a limit to the number of characters which may be typed in to an edit control.

Parameters *wParam*: *wParam* is the new maximum number of characters to be allowed. If *wParam* is zero there is no limit.

lParam: Not used.

Return value If successful, a non-zero value is returned, otherwise zero is returned.

em_LineFromChar

Returns the number of the line in an edit control which contains the specified character index.

Parameters *wParam*: *wParam* is the index of the character in the edit control or *wParam* is –1.

lParam: Not used.

Return value If *wParam* is –1 the number of the line which contains the first character in the selected text is returned, otherwise the number of the line which contains the *wParam* character index is returned.

em_LineIndex

Returns the character index of the start of a line in an edit control.

Parameters *wParam*: *wParam* is the line number. If *wParam* is –1, the line the caret currently is on is used.

lParam: Not used.

Return value The character index of the start of the line is returned.

Comments This message only applies to multiline edit controls.

em_LineLength

Returns the byte length of a line in an edit control which contains a specified character index.

Parameters *wParam*: *wParam* is the index of a character in the edit control or –1.

lParam: Not used.

Return value If *wParam* is –1 the length of the line the caret is on is returned, otherwise the length of the line which contains the *wParam* character index is returned. Any selected text, even over multiple lines, is ignored for the purposes of this message and is not included in the line length.

em_LineScroll

Scrolls an edit control.

Parameters *wParam*: Not used.

lParamLo: *lParamLo* is the number of lines to scroll vertically.

lParamHi: *lParamHi* is the number of character positions to scroll horizontally.

Return value Not used.

Comments This message only applies to multiline edit controls.

em_ReplaceSel

Replaces an edit control's selected text.

Parameters *wParam*: Not used.

lParam: *lParam* points to null terminated text to be substituted for the current selected text.

Return value Not used.

em_SetHandle

Sets an edit control's text buffer.

Parameters *wParam*: *wParam* is a local handle to a text buffer for the edit control.

lParam: Not used.

Return value Not used.

Comments The previous text buffer must be retrieved using the *em_GetHandle* message and then destroyed using the *LocalFree* function before a new text buffer is installed using this message. This message only applies to multiline edit controls.

em_SetModify

Sets an edit control's modify flag.

Parameters *wParam*: *wParam* is the new modify flag value.

lParam: Not used.

Return value Not used.

em_SetPasswordChar

Sets the character displayed instead of typed characters typed in an edit control created with the *es_Password* style.

Parameters *wParam*: *wParam* is either the new character to display or 0, in which case the actual characters typed are shown.

lParam: Not used.

Return value Not used.

em_SetRect

Sets the formatting rectangle for an edit control and re-displays the text accordingly.

Parameters *wParam*: Not used.

lParam: *lParam* points to a *TRect* structure which specifies the new formatting rectangle.

Return value Not used.

Comments This message only applies to multiline edit controls.

em_SetRectNP

Sets the formatting rectangle for an edit control without re-displaying the text.

Parameters *wParam*: Not used.

lParam: *lParam* points to a *TRect* structure which specifies the new formatting rectangle.

Return value Not used.

Comments Use this message instead of *em_SetRect* when the text will be repainted later anyway. This message only applies to multiline edit controls.

em_SetSel

Sets the selected text in an edit control.

Parameters *wParam*: Not used.

lParamLo: *lParamLo* is the starting character index.

lParamHi: *lParamHi* is the ending character index.

Return value Not used.

em_SetTabStops

Sets an edit control's tab stops.

Parameters *wParam*: *wParam* is either 1, the number of tab stops, or zero.

lParam: If *wParam* is zero and *lParam* is zero a tab stop is placed every 32 dialog units. If *wParam* is 1, a tab stop is placed at each multiple of *lParam* in dialog units. If *wParam* is neither zero nor 1, *lParam* points at an Integer array with at least *wParam* elements, each of which is greater than the one before and is the dialog unit location for that tab stop.

Return value If all tab stops were set, a non-zero value is returned, otherwise zero is returned.

Comments The current dialog unit is one-fourth of the current dialog base width unit, which can be obtained by using the *GetDialogBaseUnits* function. This message only applies to multiline edit controls.

em_SetWordBreak

Changes an edit control's word-break function.

Parameters *wParam*: Not used.

lParam: *lParam* is a procedure instance address of a word-break function. This is created by using the *MakeProcInstance* function. The word-break function would be declared like this:

```
function WordBreakFunction(EditText: PChar; CurrentWord: Integer;
EditTextCount: Integer): PChar;
```

The name *WordBreakFunction* is not literal, the function can have a different name. The *EditText* parameter points to the text of an edit control. The *CurrentWord* parameter is the index of the start of the current word in the text. *EditTextCount* parameter is the total number of bytes in the text. The word-break function must return a pointer to the character at the start of the next word in the text. If the current word is the last one, it should return a pointer to the character just after the last character in the text.

Return value Not used.

Comments The default Windows edit control word-break function defines the start of the next word to be the first non-blank character after the next series of space characters. This message only applies to multiline edit controls.

em_Undo

Undoes the last modification of an edit control's text.

Parameters *wParam*: Not used.

lParam: Not used.

Return value If successful, a non-zero value is returned, otherwise zero is returned and the edit control's text is unchanged.

Comments Each change to an edit control's text is stored in an undo buffer. One condition under which this message can fail is when there is not enough memory to create an undo buffer for the undo operation itself.

lb_AddString

Adds a string to a list box.

Parameters *wParam*: Not used.

lParam: *lParam* is a pointer to a null-terminated string to be added.

Return value If successful, the index at which the string was added is returned, otherwise *lb_ErrSpace* is returned if there wasn't enough memory available to store the string or *lb_Err* is returned if there was an error.

Comments If the list box is not sorted, the string is put at the end of the list. If the list box has the *lbs_OwnerDrawFixed* or *lbs_OwnerDrawVariable* style and does not have the *lbs_HasStrings* style, *lParam* is a 32-bit value which is stored

instead of a string and each entry added is compared to other entries one or more times via a *wm_CompareItem* message sent to the owner of the list box.

lb_DeleteString

Deletes a string from a list box.

Parameters *wParam*: *wParam* is the index of the entry to be deleted.

lParam: Not used.

Return value If *wParam* is a valid index, the number of entries remaining in the list is returned, otherwise *lb_Err* is returned.

Comments If the list box has the *lbs_OwnerDrawFixed* or *lbs_OwnerDrawVariable* style and does not have the *lbs_HasStrings* style, the associated 32-bit value is deleted and a *wm_DeleteItem* message is sent to the owner of the list box.

lb_Dir

Adds to a list box each file name in the current directory that matches a file specification and DOS file attribute.

Parameters *wParam*: *wParam* is the DOS file attribute.

lParam: *lParam* is a pointer to a null-terminated file specification string.

Return value If successful, the index of the last entry in the resulting list is returned, otherwise *lb_ErrSpace* is returned if there wasn't enough memory available to store the entries or *lb_Err* is returned if there was an error.

lb_FindString

Finds the first entry of a list box that matches a prefix string.

Parameters *wParam*: *wParam* is the index at which to start the search. The first entry looked at is the one after *wParam*. If the end of the list is reached the search will continue with entry zero until the search index reaches *wParam*. If *wParam* is –1, the entire list is searched starting at entry zero.

lParam: *lParam* is a null-terminated prefix string.

Return value If successful, the index of the first matching entry is returned, otherwise *lb_Err* is returned.

Comments If the list box has the *lbs_OwnerDrawFixed* or *lbs_OwnerDrawVariable* style and does not have the *lbs_HasStrings* style, *lParam* is a 32-bit value which is compared to each associated 32-bit value in the list.

lb_GetCount

Returns the number of entries in a list box.

Parameters *wParam*: Not used.

lParam: Not used.

Return value The count of the list box entries is returned.

lb_GetCurSel

Returns the index of the currently selected entry in a list box.

Parameters *wParam*: Not used.

lParam: Not used.

Return value If no entry is selected, *lb_Err* is returned, otherwise the index of the currently selected entry is returned.

lb_GetHorizontalExtent

Returns the width in pixels that a list box can be scrolled horizontally.

Parameters *wParam*: Not used.

lParam: Not used.

Return value The number of pixels the list box can be scrolled horizontally is returned.

Comments This message only applies to list boxes created with the *ws_HScroll* style.

lb_GetItemData

Returns the 32-bit value associated with a list box entry.

Parameters *wParam*: *wParam* is the entry index.

lParam: Not used.

Return value If successful, the 32-bit associated value is returned, otherwise *lb_Err* is returned.

lb_GetItemRect

Retrieves the bounding rectangle of a list box entry as it is displayed.

Parameters *wParam*: *wParam* is the entry index.

lParam: *lParam* points at a *TRect* structure which is to be filled in with the bounding rectangle values.

Return value If there is an error, *lb_Err* is returned.

lb_GetSel

Returns information about whether a list box entry is selected or not.

Parameters *wParam*: *wParam* is the entry index.

lParam: Not used.

Return value If there is an error, *lb_Err* is returned. If the item is selected, a positive number is returned, otherwise zero is returned.

lb_GetSelCount

Returns the number of entries currently selected in a list box.

Parameters *wParam*: Not used.

lParam: Not used.

Return value If the list box is a multi-selection list box, the number of selected entries is returned, otherwise *lb_Err* is returned .

lb_GetSelItems

Retrieves the indexes of entries currently selected in a list box.

Parameters *wParam*: *wParam* is the maximum number of entry indexes to retrieve.

lParam: *lParam* points at an Integer array large enough to hold *wParam* entry indexes.

Return value If the list box is a multi-selection list box, the indexes of up to *wParam* selected entries are placed into the *lParam* array and the total number of selected entries placed there is returned, otherwise *lb_Err* is returned .

lb_GetText

Copies a list box entry into a supplied buffer.

Parameters *wParam*: *wParam* is the entry index.

lParam: *lParam* is a pointer to a buffer. The buffer must have enough space to contain the entry and a terminating null character.

Return value Not used.

Comments If the list box has the *lbs_OwnerDrawFixed* or *lbs_OwnerDrawVariable* style and does not have the *lbs_HasStrings* style, the 32-bit value associated with the list entry is copied into the buffer.

lb_GetTextLen

Returns the length in bytes of a list box entry.

Parameters *wParam*: *wParam* is the entry index.

lParam: Not used.

Return value If *wParam* is a valid index, the length of the entry at that index is returned, otherwise *lb_Err* is returned.

lb_GetTopIndex

Returns the index of the first visible entry in a list box.

Parameters *wParam*: Not used.

lParam: Not used.

Return value The index of the first visible entry is returned.

Comments Initially the first visible entry in a list box is entry zero. If the list box is scrolled, some other entry may be at the top.

lb_InsertString

Inserts a string into a list box without sorting.

Parameters *wParam*: If *wParam* is –1, the string is added to the end of the list, otherwise *wParam* is used as the index at which to insert the string.

lParam: *lParam* points to a null terminated string to be inserted.

Return value If successful, the index at which the string was inserted is returned, otherwise *lb_ErrSpace* is returned if there wasn't enough memory available to store the string or *lb_Err* is returned if there was an error.

lb_ResetContent

Removes all entries from a list box.

Parameters *wParam*: Not used.

lParam: Not used.

Return value Not used.

Comments If the combo box has the *lbs_OwnerDrawFixed* or *lbs_OwnerDrawVariable* style and does not have the *lbs_HasStrings* style, a *wm_DeleteItem* message is sent to the owner of the list box for each entry.

lb_SelectString

Selects the first entry of a list box that matches a prefix string.

Parameters *wParam*: *wParam* is the index at which to start the search. The first entry looked at is the one after *wParam*. If the end of the list is reached the search will continue with entry zero until the search index reaches *wParam*. If *wParam* is –1, the entire list is searched starting at entry zero.

lParam: *lParam* is a null-terminated prefix string.

Return value If successful, the index of the first matching entry is returned, otherwise *lb_Err* is returned and the current selection remains unchanged.

Comments If the list box has the *lbs_OwnerDrawFixed* or *lbs_OwnerDrawVariable* style and does not have the *lbs_HasStrings* style, *lParam* is a 32-bit value which is compared to each associated 32-bit value in the list.

lb_SelItemRange

Selects or deselects consecutive items in a list box.

Parameters *wParam*: If *wParam* is zero, the entries are to be deselected, otherwise the entries are to be selected.

lParamLo: *lParamLo* is the starting entry index.

lParamHi: *lParamHi* is the ending entry index.

Return value If there is an error, *lb_Err* is returned.

Comments This message only applies to multiple-selection list boxes.

lb_SetColumnWidth

Sets the column width of a list box.

Parameters *wParam*: *wParam* is the width in pixels of each column.

lParam: Not used.

Return value Not used.

Comments This message only applies to list boxes with the *lbs_MultiColumn* style.

lb_SetCurSel

Selects a list box entry.

Parameters *wParam*: *wParam* is the entry index. If *wParam* is –1, there is to be no selection.

lParam: Not used.

Return value if *wParam* is –1 or not a valid index, *lb_Err* is returned, otherwise the index of the selected entry is returned.

lb_SetHorizontalExtent

Sets the width in pixels that a list box can be scrolled horizontally.

Parameters *wParam*: *wParam* is the number of pixels the list box can be scrolled horizontally.

lParam: Not used.

Comments This message only applies to list boxes created with the *ws_HScroll* style. The horizontal scroll bar will be enabled or disabled depending on whether the resulting extent is smaller than the width of the list box or not.

lb_SetItemData

Sets the 32-bit value associated with a list box entry.

Parameters *wParam*: *wParam* is the entry index.

lParam: *lParam* is the new 32-bit value to be associated with the entry.

Return value *lb_Err* is returned if there is an error.

lb_SetSel

Selects or deselects a list box entry.

Parameters *wParam*: If *wParam* is zero, the entry is deselected, otherwise the entry is selected.

lParam: If *lParam* is –1, this message applies to all entries in the list box, otherwise *lParamLo* is used to determine which entry to use.

lParamLo: If *lParam* is not –1, *lParamLo* is the entry index.

Return value *lb_Err* is returned if there is an error.

Comments This message only applies to multiple-selection list boxes.

lb_SetTabStops

Sets a list box's tab stops.

Parameters *wParam*: *wParam* is either 1, the number of tab stops, or zero.

lParam: If *wParam* is zero and *lParam* is zero, a tab stop is placed every 2 dialog units. If *wParam* is 1, a tab stop is placed at each multiple of *lParam* in dialog units. If *wParam* is neither zero nor 1, *lParam* points at an INT array with at least *wParam* elements, each of which is greater than the one before and is the dialog unit location for that tab stop.

Return value If all tab stops were set, a non-zero value is returned, otherwise zero is returned.

Comments The current dialog unit is one-fourth of the current dialog base width unit, which can be obtained by using the *GetDialogBaseUnits* function. This message only applies to multiple-selection list boxes.

lb_SetTopIndex

Sets the index of the first visible entry in a list box.

Parameters *wParam*: *wParam* is the entry index.

lParam: Not used.

Return value *lb_Err* is returned if there is an error.

wm_Activate

Notifies a window that it is becoming active or inactive.

Parameters *wParam*: If *wParam* is zero, the window is now inactive. If *wParam* is 1, the window is being activated by something other than a mouse click. If *wParam* is 2, the window is being activated by a mouse click.

lParamHi: *lParamHi* is non-zero if the window is minimized, otherwise it is zero.

lParamLo: If *wParam* is zero, *lParamLo* is a handle to the window being activated, otherwise *lParamLo* is a handle to the window being deactivated.

Return value Not used.

Comments If a window is not minimized and is activated, the default action taken in *DefWindowProc* is to give the window the input focus.

wm_ActivateApp

Notifies an application that a window in the application is being activated and the previously active window was in another application or that a window is being deactivated and the window becoming active is in another application.

Parameters *wParam*: If *wParam* is zero, a window in the other application is being activated, otherwise a window in the application is being activated.

lParam: *lParam* is the task handle of the other application.

Return value Not used.

wm_AskCBFormatName

Asks the clipboard owner for the clipboard data format name.

Parameters *wParam*: *wParam* is the maximum length of name that can be copied into the *lParam* buffer.

lParam: *lParam* points to a buffer into which the format name is to be copied.

Return value Not used.

Comments Messages which will be sent to the clipboard owner when the clipboard format is *cf_OwnerDisplay* are *wm_AskCBFormatName*, *wm_HScrollClipboard*, *wm_PaintClipboard*, *wm_SizeClipboard*, and *wm_VScrollClipboard*. The clipboard data and format are set by using the *SetClipboardData* function.

wm_CancelMode

Notifies an application that a message box will be displayed which cancels any mode the system is in.

Parameters *wParam*: Not used.

lParam: Not used.

Return value Not used.

Comments This message warns the application that mouse and keyboard input will be directed toward the message box. Any processing keeping track of keyboard or mouse button states and/or mouse position may be invalidated when the message box is removed.

wm_ChangeCBChain

Notifies the first window on the clipboard chain that a window is being removed from the clipboard chain.

Parameters *wParam*: *wParam* is a handle to a window which is being removed from the clipboard chain.

lParamLo: *lParamLo* is a handle to the window after the window which is being removed.

lParamHi: Not used.

Return value Not used.

Comments This message must be sent to the next window in the clipboard chain by using the *SendMessage* function. The handle of the next window in the clipboard chain is initially the value returned when a window is added to the chain using the *SetClipboardViewer* function. Whenever *wParam* is the same as this saved handle of the next window, the new handle of the next window is *lParamLo*. A window must remove itself from the clipboard chain when it receives a *wm_Destroy* message. See also *wm_DrawClipboard*.

wm_Char

Notifies the window with the focus that a non-system key was pressed.

Parameters *wParam*: *wParam* is the key value.

lParamLo: *lParamLo* is the number of times this key-press was repeated because the key was held down.

lParamHi: Bits 0-7 of *lParamHi* is the key's scan code which is OEM-dependent. Bit 8 is 1 if the key is an extended key. Bit 13 is 1 if the *Alt* key was down while the key was pressed. If the key was down before this message was sent, bit 14 is 1. Bit 15 is 1 if the key is being released, zero if it is being pressed.

Return value Not used.

Comments A non-system key is any key pressed while the *Alt* key is not pressed. *lParamLo* and bits 0-7 of *lParamHi* are generally all that an application needs. If no window has the focus, *wm_SysKeyDown*, *wm_SysChar*, and *wm_SysKeyUp* messages are sent instead of *wm_KeyDown*, *wm_Char*, and *wm_KeyUp* messages. See also *wm_DeadChar*, *wm_SysChar*, and *wm_SysDeadChar*.

wm_CharToItem

Asks the owner of a list box what the list box should do in response to a *wm_Char* message.

Parameters *wParam*: *wParam* is the key pressed.

lParamLo: *lParamLo* is a handle to the list box.

lParamHi: *lParamHi* is the current caret position.

Return value If –2 is returned, the application handled everything. If –1 is returned, the list box is to perform the default action for that key. If zero or greater is returned, the list box is to perform the default action for that key, but on the item specified by the return value. This message only applies to list boxes with the *lbs_WantKeyboardInput* style. See also *wm_VKeyToItem*.

wm_ChildActivate

Notifies a parent window that one of its child windows was moved using the *SetWindowPos* function.

Parameters *wParam*: Not used.

lParam: Not used.

Return value Not used.

wm_Clear

Deletes the current selection in a window.

Parameters *wParam*: Not used.

lParam: Not used.

Return value Not used.

wm_Close

Notifies a window that it is about to be closed.

Parameters *wParam*: Not used.

lParam: Not used.

Return value Not used.

Comments The default action taken in the *DefWindowProc* function is to call the *DestroyWindow* function to destroy the window.

wm_Command

Notifies a window that a menu item has been selected, an accelerator key has been translated, or a message has been passed to it from a child control.

Parameters *wParam*: *wParam* is the menu item, accelerator ID, or control ID.

lParamLo: If *lParamLo* is zero, the message is from a menu and *lParamHi* is not used, otherwise the value of *lParamLo* depends on *lParamHi*.

lParamHi: If *lParamHi* is 1, *lParamLo* is an accelerator ID, otherwise *lParamLo* is a handle to a child control and *lParamHi* is a notification code for a message sent to this window by the child control (See "(bn_) Button notification codes", "(en_) Edit Control notification codes", "(lbn_) List box notification codes", and "(cbn_) Combo box notification codes" in Chapter 1, "Windows styles and constants".

Return value Not used.

Comments Accelerator keys that map to System menu items, translate into *wm_SysCommand* message instead of *wm_Command* messages. *wm_Command* is sent for accelerators only if the window is not minimized or the window is minimized and the accelerator does not match any menu items on the window's menu or the System menu.

wm_Compacting

Notifies a top-level window that more than 12.5 percent of system time is being spent compacting memory.

Parameters *wParam*: *wParam* is the percentage of CPU time spent compacting memory * 65,535. For instance, if *wParam* is equal to 32,768, 50% of CPU time is being spent compacting memory.

lParam: Not used.

Return value Not used.

Comments When an application receives this message, it should free as much memory as possible. The current use of resources and the total number of running applications should be taken into account. The *GetNumTasks* function will return the number applications.

wm_CompareItem

Asks the owner of an owner-draw combo box or list box to compare two entries and return a value indicating their sort order.

Parameters *wParam*: Not used.

lParam: *lParam* points to a *TCompareItemStruct* structure. The structure contains the identifier and data for both items.

Return value –1, zero, or 1 must be returned depending on whether entry 1 sorts before, the same as, or after item 2, respectively.

Comments This message only applies to combo boxes with the *cbs_Sort* style and the *cbs_OwnerDrawFixed* or *cbs_OwnerDrawVariable* style and to list boxes with the *lbs_Sort* style and the *lbs_OwnerDrawFixed* or *lbs_OwnerDrawVariable* style.

wm_Copy

Copies the current selection to the clipboard in *cf_Text* format.

Parameters *wParam*: Not used.

lParam: Not used.

Return value Not used.

wm_Create

Notifies a window that it has been created and initialization should now be done.

Parameters *wParam*: Not used.

lParam: *lParam* points to a *TCreateStruct* structure that contains the information passed to the *CreateWindow* function.

Return value Not used.

Comments This message is sent to a window during the *CreateWindow* function call, before the window is opened.

wm_CtlColor

Gives the parent window of a child control or message box an opportunity to change the text and background colors with which the child will be drawn.

Parameters *wParam*: *wParam* is a handle to a display context for the child window.

lParamLo: *lParamLo* is a handle to the child window.

lParamHi: *lParamHi* is any of the *ctlcolor_* constants. It specifies the type of the child window. See "(ctlcolor_) Control color flags" in Chapter 1, "Windows styles and constants".

Return value Not used.

Comments The default action taken in the *DefWindowProc* function is to use the default system colors.

wm_Cut

Copies the current selection to the clipboard in *cf_Text* format and then deletes the current selection.

Parameters *wParam*: Not used.

lParam: Not used.

Return value Not used.

wm_dde_Ack

Notifies the application that it received another DDE message.

Parameters *wParam*: The handle of the window that sent the message.

lParam: If the message received was *wm_dde_Initiate*, *lParamLo* holds an atom naming the replying application and *lParamHi* holds an atom containing the topic with which the replying server window is associated. If the message received was *wm_dde_Execute*, *lParamLo* holds a record indicating the status of the response and *lParamHi* holds a handle to the data item containing the command string. For all other messages,

lParamLo holds a status record and *lParamHi* holds an atom specifying the data item for which the response is sent.

Comments This message must be sent with the *SendMessage* function. The first parameter should be the handle of the window to receive the message.

wm_dde_Advise

Sent by a client application requesting that the server (receiving) application supply an update whenever the data item changes.

Parameters *wParam*: The handle of the sending window.

lParamLo: A *TDDEAdvise* record indicating how to send the data.

lParamHi: An atom specifying the data item requested.

Comments This message must be sent with the *PostMessage* function. The first parameter should be the handle of the window to receive the message.

wm_dde_Data

Sent by a server application to transmit a data item value or to notify the client that the item is available.

Parameters *wParam*: The handle of the sending window.

lParamLo: A handle to the global memory block containing the data, stored in a *TDDEData* record, or zero if the message is simply a notification of change.

lParamHi: An atom specifying what data item was sent.

Comments This message must be sent with the *PostMessage* function. The first parameter should be the handle of the window to receive the message.

wm_dde_Execute

Sent by a client application to transmit a series of commands to be processed by the server application.

Parameters *wParam*: The handle of the sending window.

lParamLo: Reserved.

lParamHi: A handle to a global memory object containing the commands.

Comments This message must be sent with the *PostMessage* function. The first parameter should be the handle of the window to receive the message.

wm_dde_Initiate

Sent by either the client or server to initiate a conversation. Responding applications are expected to send a *wm_dde_Ack* message.

Parameters *wParam*: The handle of the sending window.

lParamLo: An atom specifying the name of the application with which a conversation is requested, or zero for a conversation with any application.

lParamHi: An atom specifying the topic about which the conversation is requested, or zero for a conversation about any topic.

Comments This message must be sent with the *SendMessage* function. The first parameter should be the handle of the window to receive the message.

wm_dde_Poke

Sent by the client application to ask the server application to receive unsolicited data. The server responds with a *wm_dde_Ack* message.

Parameters *wParam*: The handle of the sending window.

lParamLo: A handle to a *TDDEPoke* record.

lParamHi: An atom identifying the data item.

Comments This message must be sent with the *PostMessage* function. The first parameter should be the handle of the window to receive the message.

wm_dde_Request

Sent by the client application to request the value of a particular data item.

Parameters *wParam*: The handle of the sending window.

lParamLo: A clipboard format number. See (cf_) Clipboard formats in Chapter 1, "Windows styles and constants."

lParamHi: An atom identifying the requested data item.

Comments This message must be sent with the *PostMessage* function. The first parameter should be the handle of the window to receive the message.

wm_dde_Terminate

Sent by any application to terminate a conversation.

Parameters *wParam*: The handle of the sending window.

lParam: Reserved.

Comments This message must be sent with the *PostMessage* function. The first parameter should be the handle of the window to receive the message.

wm_dde_Unadvise

Sent by a client application to a server application to inform the server that it no longer needs to update a particular item or clipboard format for the item.

Parameters *wParam*: The handle of the sending window.

lParamLo: A clipboard format number. See (cf_) Clipboard formats in Chapter 1, "Windows styles and constants."

lParamHi: An atom identifying the data item.

Comments This message must be sent with the *PostMessage* function. The first parameter should be the handle of the window to receive the message.

wm_DeadChar

Notifies a window of a dead character.

Parameters *wParam*: *wParam* is the key value.

lParamLo: *lParamLo* is the number of times this key-press has been repeated because the key is being held down.

lParamHi: Bits 0-7 of *lParamHi* is the key's scan code which is OEM-dependent. Bit 8 is 1 if the key is an extended key. Bit 13 is 1 if the *Alt* key is held down while the key is pressed. If the key was already down before this message was sent, bit 14 is 1 and the repeat count will be a positive number. Bit 15 is 1 if the key is being released, zero if it is being pressed.

Return value Not used.

Comments Dead keys include the umlaut character and accents. This message can be used to give feedback for keys pressed that do not necessarily result in a character by themselves. *lParamLo* and bits 0-7 of *lParamHi* are generally all that an application needs. See also *wm_Char*, *wm_SysChar*, and *wm_SysDeadChar*.

wm_DeleteItem

Notifies the owner of a list box or combo box that a list box entry has been deleted.

Parameters *wParam*: Not used.

lParam: Not used.

Return value Not used.

Comments This message only applies to combo boxes with the *cbs_OwnerDrawFixed* or *cbs_OwnerDrawVariable* style and list boxes with the *lbs_OwnerDrawFixed* or *lbs_OwnerDrawVariable* style. This message is sent when the combo box or list box is destroyed or the entry is deleted by a *lb_DeleteString*, *lb_ResetContent*, *cb_DeleteString*, or *cb_ResetContent* message.

wm_Destroy

Notifies a window that it is going to be destroyed.

Parameters *wParam*: Not used.

lParam: Not used.

Return value Not used.

Comments Any window which is on the clipboard chain must remove itself from the clipboard chain by using the *ChangeClipboardChain* function before it returns from the *wm_Destroy* message. This message is sent from the *DestroyWindow* function after the window is removed from the screen. A window receives this message before any of its child windows are destroyed.

wm_DestroyClipboard

Notifies a clipboard owner that the clipboard has been emptied by an *EmptyClipboard* function call.

Parameters *wParam*: Not used.

lParam: Not used.

Return value Not used.

wm_DevModeChange

Notifies each top-level window that the device-mode settings have changed.

Parameters *wParam*: Not used.

lParam: *lParam* points to the device name.

Return value Not used.

Comments The device name is the string found in the Windows initialization file, WIN.INI.

wm_DrawClipboard

Notifies the first window on the clipboard chain that the clipboard contents have changed.

Parameters *wParam*: Not used.

lParam: Not used.

Return value Not used.

Comments This message must be sent to the next window in the clipboard chain by using the *SendMessage* function. The handle of the next window in the clipboard chain is initially the value returned when a window is added to the chain by the *SetClipboardViewer* function. New values for this handle are sent via *wm_ChangeCBChain* messages. A window must remove itself from the clipboard chain when it receives a *wm_Destroy* message.

wm_DrawItem

Informs an owner-draw button, combo box, list box, or menu that it must be redrawn.

Parameters *wParam*: Not used.

lParam: Points to a *TDrawItemStruct* structure which contains information on the item and the drawing operation to be done.

Return value Not used.

Comments All objects selected for the display context found in the *TDrawItemStruct* structure must be restored before returning from this message.

wm_Enable

Notifies a window when it is enabled or disabled.

Parameters *wParam*: If *wParam* is zero, the window has been disabled, otherwise it has been enabled.

lParam: Not used.

Return value Not used.

wm_EndSession

Tells an application that has responded nonzero to a *wm_QueryEndSession* message whether the session is being ended.

Parameters *wParam*: *wParam* is zero, the session is not being ended, otherwise the session is being ended.

lParam: Not used.

Return value Not used.

Comments If *wParam* is not zero, an application should perform all tasks required for termination before returning from this message since Windows might terminate any time after all applications have returned from handling this message. The application does not need to call the either the *DestroyWindow* or *PostQuitMessage* function as when handling a *wm_Destroy* message.

wm_EnterIdle

Notifies a main window that the system is idle due to displaying a modal dialog or a menu.

Parameters *wParam*: If the system is idle because a dialog box is being displayed, *wParam* will be *msgf_DialogBox*. If the system is idle because a menu is being displayed, *wParam* will be *msgf_Menu*.

lParamLo: *lParamLo* is a handle to the dialog box or menu when *wParam* is *msgf_DialogBox* or *msgf_Menu* respectively.

lParamHi: Not used.

Return value Not used.

Comments The system becomes idle when a modal dialog box or displayed menu has no messages queued after processing at least one previous message. The default value returned by *DefWindowProc* is zero.

wm_EraseBkgnd

Notifies a window that its background must be erased to prepare for painting an invalidated region.

Parameters *wParam*: *wParam* is a handle to the device-context.

lParam: Not used.

Return value If an application handles this message and erases the window background, it should return a non-zero value, otherwise it should return zero.

Comments The default action taken in the *DefWindowProc* function is to erase the background using the class background brush from the class structure. If the class background brush is 0, the application should align the origin of the brush to be used with the window by calling the *UnrealizeObject* function for the brush, select the brush, and then erase the background using the brush. Windows assumes the *mm_Text* mapping mode. If the device context uses a different mapping mode, the area erased might extend outside the visible portion of the client area.

wm_FontChange

Notifies a top-level window that the pool of font resources has changed.

Parameters *wParam*: Not used.

lParam: Not used.

Return value Not used.

Comments Any application that adds or removes fonts from the system should send this message to every top-level window by using the *SendMessage* function. The *AddFontResource* function is used to add fonts to the system. The *RemoveFontResource* function is used to remove them.

wm_GetDlgCode

Allows an application to override the handling of direction keys and the *Tab* key in a control.

Parameters *wParam*: Not used.

lParam: Not used.

Return value The application must return a value composed of any *dlgc_* constants bit-or'ed together depending on which inputs the application wants to process. See "(dlgc_) Dialog codes" in Chapter 1, "Windows styles and constants".

Comments The default value returned by *DefWindowProc* is zero. The window functions for the predefined control classes may return a code which is not zero. This message and non-default return values from it are useful only with user-defined dialog controls or subclassed standard controls.

wm_GetFont

Returns a dialog box's current font.

Parameters *wParam*: Not used.

lParam: Not used.

Return value The return value is 0 if the dialog box is using the system font, otherwise it is a handle to the font being used. See also *wm_SetFont*.

wm_GetMinMaxInfo

Allows a window to change its default maximized size, its default maximized position, or its default minimum and maximum tracking sizes.

Parameters *wParam*: Not used.

lParam: *lParam* points at an array of 5 *Point* structures. *lParam*[0] is used internally by Windows. *lParam*[1] is the maximized size. *lParam*[2] is the position of the upper-left corner of the window when it is maximized. *lParam*[3] is the minimum tracking size of the window. *lParam*[4] is the maximum tracking size of the window.

Return value Elements 1 to 4 of the *lParam* array may be modified as desired.

Comments The tracking sizes are the minimum and maximum sizes allowed when resizing the window. This message gives the application an opportunity to change the default sizes before Windows uses them.

wm_GetText

Copies a window's associated text into a supplied buffer.

Parameters *wParam*: *wParam* is the maximum number of bytes that may be copied into the *lParam* buffer.

lParam: *lParam* is a pointer to a buffer. The buffer must be at least *wParam* bytes long.

Return value If the window is a list box and no item is selected, *lb_Err* is returned. If the window is a combo box which has no edit control, *cb_Err* is returned. Otherwise, the number of bytes copied, including a null terminating character, is returned.

Comments For edit controls, the text is the contents of the edit control. For button controls, the text is the button name. For list boxes, the text is the currently selected item, if any. For combo boxes, the text is the contents of the combo box's edit control. For all other windows, the text is the window caption. See also *wm_GetTextLength* and *wm_SetText*.

wm_GetTextLength

Returns the length, in bytes, of a window's associated text.

Parameters *wParam*: Not used.

lParam: Not used.

Return value The length of the associated text, not including a null terminating character, is returned.

Comments For edit controls, the text is the contents of the edit control. For button controls, the text is the button name. For list boxes, the text is the currently selected item, if any. For combo boxes, the text is the contents of the combo box's edit control. For all other windows, the text is the window caption. See also *wm_GetText*.

wm_HScroll

Notifies a window that a horizontal scroll bar has been clicked.

Parameters *wParam*: *wParam* is a scroll bar code which describes the effect of the scroll bar click. It may be any of the *sb_* constants that apply to horizontal scroll bar controls. See "(sb_) Scroll bar commands" in Chapter 1, "Windows styles and constants".

lParamLo: Not used.

lParamHi: *lParamHi* is the handle of the scroll bar control. *lParamHi* will be zero if the scroll bar control is a control created with the window by using the *ws_HScroll* style.

Return value Not used.

Comments If an application scrolls the text in a window, it must also use the *SetScrollPos* function to reset the scroll bar's thumb position.

wm_HScrollClipboard

Notifies the owner of a clipboard with the *cf_OwnerDisplay* format that the horizontal scroll bar in the clipboard application has been clicked.

Parameters *wParam*: *wParam* is a handle to the clipboard application window.

lParamLo: *lParamLo* is a scroll bar code which describes the effect of the scroll bar click. It may be any of the *sb_* constants that apply to horizontal scroll bar controls. See "(sb_) Scroll bar commands" in Chapter 1, "Windows styles and constants".

lParamHi: Not used.

Return value Not used.

Comments The clipboard owner must repaint the clipboard application window or use the *InvalidateRect* function. The clipboard application window scroll bar position must be reset using the *SetScrollPos* function. Messages which will be sent to the clipboard owner when the clipboard format is *cf_OwnerDisplay* are *wm_AskCBFormatName*, *wm_HScrollClipboard*, *wm_PaintClipboard*, *wm_SizeClipboard*, and *wm_VScrollClipboard*. The clipboard data and format are set by using the *SetClipboardData* function.

wm_IconEraseBkgnd

Notifies a minimized window that its background must be filled to prepare for painting the icon.

Parameters *wParam*: *wParam* is the device-context of the icon.

lParam: Not used.

Return value Not used.

Comments This message only applies to minimized (iconic) windows that have a class icon defined for them. Other windows receive the *wm_EraseBkgnd* message instead. The default action taken in the *DefWindowProc* function is to fill the icon background using the background brush of the parent window.

wm_InitDialog

Notifies an application that a dialog box is about to be displayed and should be initialized.

Parameters *wParam*: *wParam* is the ID of the first control in the dialog box that can have the input focus.

lParam: *lParam* is the *InitParam* value passed to the function that created the dialog. Functions that have this parameter are *CreateDialogIndirectParam*, *CreateDialogParam*, *DialogBoxIndirectParam*, and *DialogBoxParam*. If the dialog was created using *CreateDialog*, *CreateDialogIndirect*, *DialogBox*, or *DialogBoxIndirect*, *lParam* is zero.

Return value If the application sets the input focus to one of the dialog's controls, it may return zero, otherwise it must return a non-zero value.

Comments This message allows an application to initialize a dialog and set the input focus to any control in the dialog just before the dialog is displayed. If the return value is zero, Windows will set the input focus to the control identified by *wParam*. *wParam* will usually be the ID of the first item in the dialog box with the *ws_TabStop* style. An example of other initialization best done at this time is to set the dialog control's font using the *wm_SetFont* message.

wm_InitMenu

Notifies an application that a menu is about to be displayed.

Parameters *wParam*: *wParam* is a handle to the menu.

lParam: Not used.

Return value Not used.

Comments This message is sent when the mouse is clicked on the menu bar or a menu key is pressed. It gives the application the opportunity to change the state of any menu items before the menu is displayed.

wm_InitMenuPopup

Notifies an application that a pop-up menu is about to be displayed.

Parameters *wParam*: *wParam* is a handle to the pop-up menu.

lParamLo: *lParamLo* is the index of the pop-up menu in the main menu.

lParamHi: *lParamHi* is non-zero if the pop-up menu is the system menu, otherwise it is zero.

Return value Not used.

Comments This message gives the application the opportunity to change the state of any menu items before the pop-up menu is displayed.

wm_KeyDown

Notifies the window with the focus that a non-system key was pressed.

Parameters *wParam*: *wParam* is the virtual-key code.

lParamLo: *lParamLo* is the number of times this key-press was repeated because the key is being held down.

lParamHi: Bits 0-7 of *lParamHi* is the key's scan code which is OEM-dependent. Bit 8 is 1 if the key is an extended key. Bit 13 is 1 if the *Alt* key was down while the key was pressed. If the key was down before this message was sent, bit 14 is 1. Bit 15 is 1 if the key is being released, zero if it is being pressed.

Return value Not used.

Comments A non-system key is any key pressed while the *Alt* key is not pressed. For this message bits 13 and 15 of *lParamHi* will be zero. Due to auto-repeat, more than one *wm_KeyDown* message may be sent before a *wm_KeyUp* message is sent. If no window has the focus, *wm_SysKeyDown*, *wm_SysChar*, and *wm_SysKeyUp* messages are sent instead of *wm_KeyDown*, *wm_Char*, and *wm_KeyUp* messages.

wm_KeyUp

Notifies the window with the focus that a non-system key was released.

Parameters *wParam*: *wParam* is the virtual-key code.

lParamLo: *lParamLo* is the number of times this key-press was repeated because the key was held down.

lParamHi: Bits 0-7 of *lParamHi* is the key's scan code which is OEM-dependent. Bit 8 is 1 if the key is an extended key. Bit 13 is 1 if the *Alt* key was down while the key was pressed. If the key was down before this message was sent, bit 14 is 1. Bit 15 is 1 if the key is being released, zero if it is being pressed.

Return value Not used.

Comments A non-system key is any key pressed while the *Alt* key is not pressed. For this message bits 13 *lParamHi* will be zero and bit 15 of *lParamHi* will be 1. Due to auto-repeat, more than one *wm_KeyDown* message may be sent before a *wm_KeyUp* message is sent. If no window has the focus, *wm_SysKeyDown*, *wm_SysChar*, and *wm_SysKeyUp* messages are sent instead of *wm_KeyDown*, *wm_Char*, and *wm_KeyUp* messages.

wm_KillFocus

Notifies a window that it is about to lose the focus.

Parameters *wParam*: *wParam* is the handle of the window that will be receiving the focus.

lParam: Not used.

Return value Not used.

Comments Any displayed caret should be destroyed at this time. *wParam* may be zero.

wm_LButtonDblClk

Notifies a window that the left mouse button was double-clicked.

Parameters *wParam*: *wParam* is a value which indicates which virtual keys are down. It is a combination of the *mk_* constants. See "(mk_) Key state masks" in Chapter 1, "Windows styles and constants".

lParamLo: *lParamLo* is the x-coordinate of the mouse cursor.

lParamHi: *lParamHi* is the y-coordinate of the mouse cursor.

Return value Not used.

Comments This message only applies to windows created with the *cs_DblClks* style. The mouse coordinates are relative to the upper left corner of the window. A double-click is actually two single-clicks within the system's double-click time limit. A double click results in a down-click message, an up-click message, a double-click message, and a second up-click message. See also *wm_LButtonDown*, *wm_LButtonUp*, *wm_MButtonDblClk*, *wm_MButtonDown*, *wm_MButtonUp*, *wm_MouseMove*, *wm_NCLButtonDblClk*, *wm_NCLButtonDown*, *wm_NCLButtonUp*, *wm_NCMButtonDblClk*, *wm_NCMButtonDown*, *wm_NCMButtonUp*, *wm_NCMouseMove*, *wm_NCRButtonDblClk*, *wm_NCRButtonDown*, *wm_NCRButtonUp*, *wm_RButtonDblClk*, *wm_RButtonDown*, and *wm_RButtonUp*.

wm_LButtonDown

Notifies a window that the left mouse button was clicked.

Parameters *wParam*: *wParam* is a value which indicates which virtual keys are down. It is a combination of the *mk_* constants. See "(mk_) Key state masks" in Chapter 1, "Windows styles and constants".

lParamLo: *lParamLo* is the x-coordinate of the mouse cursor.

lParamHi: *lParamHi* is the y-coordinate of the mouse cursor.

Return value Not used.

Comments The mouse coordinates are relative to the upper left corner of the window. See also *wm_LButtonDblClk*, *wm_LButtonUp*, *wm_MButtonDblClk*, *wm_MButtonDown*, *wm_MButtonUp*, *wm_MouseMove*, *wm_NCLButtonDblClk*, *wm_NCLButtonDown*, *wm_NCLButtonUp*, *wm_NCMButtonDblClk*, *wm_NCMButtonDown*, *wm_NCMButtonUp*,

wm_NCMouseMove, *wm_NCRButtonDblClk*, *wm_NCRButtonDown*, *wm_NCRButtonUp*, *wm_RButtonDblClk*, *wm_RButtonDown*, and *wm_RButtonUp*.

wm_LButtonUp

Notifies a window that the left mouse button was released.

Parameters *wParam*: *wParam* is a value which indicates which virtual keys are down. It is a combination of the *mk_* constants. See "(mk_) Key state masks" in Chapter 1, "Windows styles and constants".

lParamLo: *lParamLo* is the x-coordinate of the mouse cursor.

lParamHi: *lParamHi* is the y-coordinate of the mouse cursor.

Return value Not used.

Comments The mouse coordinates are relative to the upper left corner of the window. See also *wm_LButtonDblClk*, *wm_LButtonDown*, *wm_MButtonDblClk*, *wm_MButtonDown*, *wm_MButtonUp*, *wm_MouseMove*, *wm_NCLButtonDblClk*, *wm_NCLButtonDown*, *wm_NCLButtonUp*, *wm_NCMButtonDblClk*, *wm_NCMButtonDown*, *wm_NCMButtonUp*, *wm_NCMouseMove*, *wm_NCRButtonDblClk*, *wm_NCRButtonDown*, *wm_NCRButtonUp*, *wm_RButtonDblClk*, *wm_RButtonDown*, and *wm_RButtonUp*.

wm_MButtonDblClk

Notifies a window that the middle mouse button was double-clicked.

Parameters *wParam*: *wParam* is a value which indicates which virtual keys are down. It is a combination of the *mk_* constants. See "(MK_) Key state masks" in Chapter 1, "Windows styles and constants".

lParamLo: *lParamLo* is the x-coordinate of the mouse cursor.

lParamHi: *lParamHi* is the y-coordinate of the mouse cursor.

Return value Not used.

Comments This message only applies to windows created with the *cs_DblClks* style. The mouse coordinates are relative to the upper left corner of the window. A double-click is actually two single-clicks within the system's double-click time limit. A double click results in a down-click message, an up-

click message, a double-click message, and a second up-click message. See also *wm_LButtonDblClk*, *wm_LButtonDown*, *wm_LButtonUp*, *wm_MButtonDown*, *wm_MButtonUp*, *wm_MouseMove*, *wm_NCLButtonDblClk*, *wm_NCLButtonDown*, *wm_NCLButtonUp*, *wm_NCMButtonDblClk*, *wm_NCMButtonDown*, *wm_NCMButtonUp*, *wm_NCMouseMove*, *wm_NCRButtonDblClk*, *wm_NCRButtonDown*, *wm_NCRButtonUp*, *wm_RButtonDblClk*, *wm_RButtonDown*, and *wm_RButtonUp*.

wm_MButtonDown

Notifies a window that the middle mouse button was clicked.

Parameters *wParam*: *wParam* is a value which indicates which virtual keys are down. It is a combination of the *mk_* constants. See "(mk_) Key state masks" in Chapter 1, "Windows styles and constants".

lParamLo: *lParamLo* is the x-coordinate of the mouse cursor.

lParamHi: *lParamHi* is the y-coordinate of the mouse cursor.

Return value Not used.

Comments The mouse coordinates are relative to the upper left corner of the window. See also *wm_LButtonDblClk*, *wm_LButtonDown*, *wm_LButtonUp*, *wm_MButtonDblClk*, *wm_MButtonUp*, *wm_MouseMove*, *wm_NCLButtonDblClk*, *wm_NCLButtonDown*, *wm_NCLButtonUp*, *wm_NCMButtonDblClk*, *wm_NCMButtonDown*, *wm_NCMButtonUp*, *wm_NCMouseMove*, *wm_NCRButtonDblClk*, *wm_NCRButtonDown*, *wm_NCRButtonUp*, *wm_RButtonDblClk*, *wm_RButtonDown*, and *wm_RButtonUp*.

wm_MButtonUp

Notifies a window that the middle mouse button was released.

Parameters *wParam*: *wParam* is a value which indicates which virtual keys are down. It is a combination of the *mk_* constants. See "(mk_) Key state masks" in Chapter 1, "Windows styles and constants".

lParamLo: *lParamLo* is the x-coordinate of the mouse cursor.

lParamHi: *lParamHi* is the y-coordinate of the mouse cursor.

Return value Not used.

Comments The mouse coordinates are relative to the upper left corner of the window. See also *wm_LButtonDblClk*, *wm_LButtonDown*, *wm_LButtonUp*, *wm_MButtonDblClk*, *wm_MButtonDown*, *wm_MouseMove*, *wm_NCLButtonDblClk*, *wm_NCLButtonDown*, *wm_NCLButtonUp*, *wm_NCMButtonDblClk*, *wm_NCMButtonDown*, *wm_NCMButtonUp*, *wm_NCMouseMove*, *wm_NCRButtonDblClk*, *wm_NCRButtonDown*, *wm_NCRButtonUp*, *wm_RButtonDblClk*, *wm_RButtonDown*, and *wm_RButtonUp*.

wm_MDIActivate

Tells an multiple document interface (MDI) client window to activate a different MDI child window. This message is then passes to the MDI child windows being deactivated and activated.

Parameters *wParam*: The MDI client window does not use *wParam*. For an MDI child window if *wParam* is zero, the window is being deactivated, otherwise the window is being activated.

lParamLo: *lParamLo* is a handle to the MDI child window to be activated.

lParamHi: *lParamHi* is a handle to the MDI child window to be deactivated.

Return value Not used.

Comments When an MDI client window receives this message, it must send a *wm_MDIActivate* message with the appropriate *wParam* parameter to both the MDI child window being deactivated and the MDI child window being activated. When an MDI frame window becomes active, the MDI child window that last received a *wm_MDIActivate* message with a non-zero *wParam* receives a *wm_NCActivate* message but does not receive another *wm_MDIActivate* message. If the MDI child window being deactivated is maximized, it will be restored and the MDI child window being activated will be maximized.

wm_MDICascade

Arranges a multiple document interface (MDI) client window's MDI child windows in a cascade format.

Parameters *wParam*: Not used.

lParam: Not used.

Return value Not used.

wm_MDICreate

Creates a multiple document interface (MDI) child window for an MDI client window.

Parameters *wParam*: Not used.

lParam: *lParam* points at an *TMDICreateStruct* structure.

Return value The high-order word of the return value is zero. The low-order word of the return value is the ID of the new MDI child window.

Comments An MDI child window will be created with the *ws_Child*, *ws_ClipSiblings*, *ws_ClipChildren*, *ws_SysMenu*, *ws_Caption*, *ws_ThickFrame*, *ws_MinimizeBox*, and *ws_MaximizeBox* styles along with any additional styles found in the *lParam TMDICreateStruct* structure. The title of the MDI child window is added to the window menu of the MDI frame window. All child windows of the client window should be created using this message. When an MDI child window is created, the *wm_Create* message is sent to it. with a *lParam* parameter pointing at a *TCreateStruct* structure which has a field which points at the *TMDICreateStruct* structure sent to the *wm_MDICreate* message that created the MDI child window. This message is not reentrant, for instance a *wm_MDICreate* message should not be sent while an MDI child window is handling its *wm_Create* message.

wm_MDIDestroy

Asks a multiple document interface (MDI) client window to close an MDI child window.

Parameters *wParam*: *wParam* is a handle to the MDI child window.

lParam: Not used.

Return value Not used.

Comments The title of the MDI child window is removed from the frame window and the MDI child window is deactivated. All MDI child windows should be closed with this message.

wm_MDIGetActive

Returns the active multiple document interface (MDI) child window and information about whether or not it is maximized.

Parameters *wParam*: Not used.

lParam: Not used.

Return value The low-order word of the return value is a handle to the MDI child window which is active. If the MDI child window is maximized, the high-order word is 1, otherwise, the high-order word is zero.

wm_MDIIconArrange

Arranges a multiple document interface (MDI) client window's minimized MDI child windows.

Parameters *wParam*: Not used.

lParam: Not used.

Return value Not used.

Comments MDI child windows that are not in iconic form are not affected.

wm_MDIMaximize

Asks a multiple document interface (MDI) client window to maximize an MDI child window.

Parameters *wParam*: *wParam* is the MDI child window ID.

lParam: Not used.

Return value Not used.

Comments A maximized MDI child window's client fills the MDI client window's client area, the maximized MDI child window's system menu is placed in the MDI frame window's menu bar, and the MDI child window's title is added to the MDI frame window's title.

wm_MDINext

Activates the next multiple document interface (MDI) child window.

Parameters *wParam*: Not used.

lParam: Not used.

Return value Not used.

Comments The next MDI child window is the one immediately behind the currently active MDI child window. The currently active MDI child window is placed behind all other MDI child windows.

wm_MDIRestore

Restores a maximized or minimized multiple document interface (MDI) child window.

Parameters *wParam*: *wParam* is the MDI child window's ID.

lParam: Not used.

Return value Not used.

wm_MDISetMenu

Replaces the menu and/or the Window pop-up menu of a multiple document interface (MDI) frame window.

Parameters *wParam*: Not used.

lParamLo: *lParamLo* is a handle to a new MDI frame window menu or *NULL*.

lParamHi: *lParamHi* is a handle to a new Window pop-up men or *NULL*.

Return value The handle of the old MDI frame window menu is returned.

Comments If either *lParamLo* or *lParamHi* are *NULL*, the corresponding menu is not changed. An application must use the *DrawMenuBar* function to update the menu bar after sending this message. MDI child window menu items from the old Window pop-up menu are removed and placed in the new Window pop-up menu. The System menu and restore controls for a maximized MDI child window are removed from the old MDI frame window menu and added to the new MDI frame window menu.

wm_MDITile

Arranges a multiple document interface (MDI) client window's MDI child windows in a tiled format.

Parameters *wParam*: Not used.

lParam: Not used.

Return value Not used.

wm_MeasureItem

Asks the owner of an owner-draw button, combo box, list box, or menu for that control's dimensions.

Parameters *wParam*: Not used.

lParam: *lParam* points to a *TMeasureItemStruct* structure.

Return value Not used.

Comments This message only applies to buttons with the *bs_OwnerDraw* style, combo boxes with the *cbs_OwnerDrawFixed* or *cbs_OwnerDrawVariable* style, list boxes with the *lbs_OwnerDrawFixed* or *lbs_OwnerDrawVariable* style, and owner-draw menu items. This message is sent to a control's owner when the control is created. The *TMeasureItemStruct* structure pointed to by *lParam* must be filled in with the correct values for the control. For list boxes with the *lbs_OwnerDrawVariable* style and combo boxes with the *cbs_OwnerDrawVariable* style, this message is sent once for each item, otherwise this message is sent once for each control or menu item. If a dialog owns a combo box with the *cbs_OwnerDrawFixed* style or a list box with the *lbs_OwnerDrawFixed* style, it will receive the *wm_MeasureItem* message before the *wm_InitDialog* message.

wm_MenuChar

Notifies the owner of the current menu that an undefined menu mnemonic character has been pressed.

Parameters *wParam*: *wParam* is the ASCII value of the character.

lParamLo: *lParamLo* is *mf_Popup* if the menu is a popup menu, or *mf_SysMenu* if the menu is a system menu.

lParamHi: *lParamHi* is a handle to the current menu.

Return value If the high-order word of the return value is 0, Windows will discard the character and send a beep to the speaker. If the high-order word of the return value is 1, Windows will close the current menu. If the high-order word of the return value is 2, Windows will select the menu item that is in the low-order word of the return value.

Comments Application that use accelerators to select bitmaps placed in a menu should handle this message.

wm_MenuSelect

Notifies the owner of a menu that a menu item has been selected.

Parameters *wParam*: *wParam* is the menu item ID or the handle of a pop-up menu.

lParamLo: *lParam* LO is either –1 or a combination of the *mf_Bitmap*, *mf_Checked*, *mf_Disabled*, *mf_Grayed*, *mf_MouseSelect*, *mf_OwnerDraw*,

mf_Popup, and *mf_SysMenu* flags. See "(mf_) Menu flags" in Chapter 1, "Windows styles and constants".

lParamHi: *lParamHi* is zero if the menu is the system menu or *lParam* is –1, otherwise *lParamHi* is a handle to the menu.

Return value Not used.

Comments if *lParamLo* is –1 and *lParamHi* is zero, the menu has been closed because the mouse was clicked outside the menu or the *Esc* key was pressed.

wm_MouseActivate

Notifies an inactive window that the mouse was clicked over it.

Parameters *wParam*: *wParam* is a handle to the topmost parent of the window.

lParamLo: *lParamLo* is one of the *ht* constants. See "(ht) Hit test codes" in Chapter 1, "Windows styles and constants". These are the same values returned by the *wm_NCHitTest* message.

lParamHi: *lParamHi* is the mouse message number.

Return value If the return value is *ma_Activate*, the first window that received this message will be activated. If the return value is *ma_NoActivate*, the first window that received this message will not be activated. If the return value is *ma_ActivateAndEat*, the first window that received this message will be activated and the mouse event will be discarded. No other return values are legal.

Comments Each window may pass this message to *DefWindowProc*, which passes it to the window's parent window. If the return value is non-zero at any point, this process stops and the window is not activated. See also *wm_NCHitTest*.

wm_MouseMove

Notifies a window that the mouse has moved while over the window's client area.

Parameters *wParam*: *wParam* is a value which indicates which virtual keys are down. It is a combination of the *mk_* constants. See "(mk_) Key state masks" in Chapter 1, "Windows styles and constants".

lParamLo: *lParamLo* is the x-coordinate of the mouse cursor.

lParamHi: *lParamHi* is the y-coordinate of the mouse cursor.

Return value Not used.

Comments The mouse coordinates are relative to the upper left corner of the window. See also *wm_LButtonDblClk*, *wm_LButtonDown*, *wm_LButtonUp*, *wm_MButtonDblClk*, *wm_MButtonDown*, *wm_MButtonUp*, *wm_NCLButtonDblClk*, *wm_NCLButtonDown*, *wm_NCLButtonUp*, *wm_NCMButtonDblClk*, *wm_NCMButtonDown*, *wm_NCMButtonUp*, *wm_NCMouseMove*, *wm_NCRButtonDblClk*, *wm_NCRButtonDown*, *wm_NCRButtonUp*, *wm_RButtonDblClk*, *wm_RButtonDown*, and *wm_RButtonUp*.

wm_Move

Notifies a window that it has been moved.

Parameters *wParam*: Not used.

lParamLo: *lParamLo* is the new x-coordinate of the upper-left corner of the window's client area.

lParamHi: *lParamHi* is the new y-coordinate of the upper-left corner of the window's client area.

Return value Not used.

Comments For pop-up and overlapped windows, the new coordinates are screen relative. For child windows the new coordinates are relative to the parent window's client area.

wm_NCActivate

Notifies a window that its caption bar or icon needs to change to show an active or inactive state.

Parameters *wParam*: If *wParam* is zero, the window is being inactivated, otherwise it is being activated.

lParam: Not used.

Return value Not used.

Comments The default action taken in *DefWindowProc* is to draw a gray caption for an inactive window or a black caption for an active window. At this point, no distinction is made between active and inactive icons.

wm_NCCalcSize

Asks a window for the size of the windows client area.

Parameters *wParam*: Not used.

lParam: *lParam* points to a *TRect* structure which contains the screen coordinates of the window.

Return value Not used.

Comments The area specified in *lParam* includes the client area, borders, caption, and scroll bars. The default action taken in *DefWindowProc* is to calculate the size of the client area. The calculation takes any borders, caption, and scroll bars into account. The resulting size of the client area is placed in the *lParam TRect* structure.

wm_NCCreate

Notifies an application that a window is just beginning to be created.

Parameters *wParam*: *wParam* is a handle to the window being created.

lParam: *lParam* points at the window's *TCreateStruct* structure.

Return value If the window is successfully created, a non-zero value is returned, otherwise zero is returned.

Comments If this message returns zero, the *CreateWindow* function (or other function used to create a window) will also return zero. The default action taken by *DefWindowProc* is to initialize the window's scroll bars, set the window text, and allocate memory for internal data structures.

wm_NCDestroy

Notifies a window that its non-client area is being destroyed.

Parameters *wParam*: Not used.

lParam: Not used.

Return value Not used.

Comments This message is sent by the *DestroyWindow* function after it sends the *wm_Destroy* message. The default action taken by *DefWindowProc* is to free any memory associated with the window.

wm_NCHitTest

Notifies the window receiving the mouse input that the mouse has moved.

Parameters *wParam*: Not used.

lParamLo: *lParamLo* is the x-coordinate of the mouse cursor.

lParamHi: *lParamHi* is the y-coordinate of the mouse cursor.

Return value The default return value, if this message is passed to *DefWindowProc*, will be one of the *ht* constants: *htBottom, htBottomLeft, htBottomRight, htCaption, htClient, htError, htGrowBox, htHScroll, htLeft, htMenu, htNowhere, htReduce, htRight, htSize, htSysMenu, htTop, htTopLeft, htTopRight, htTransparent, htVScroll,* or *htZoom*. See "(ht) Hit test codes" in Chapter 1, "Windows styles and constants".

Comments The mouse coordinates are relative to the upper left corner of the screen. Normally the window that contains the mouse cursor receives all mouse input, but this can be overridden by using the GetCapture function. See also *wm_LButtonDblClk, wm_LButtonDown, wm_LButtonUp, wm_MButtonDblClk, wm_MButtonDown, wm_MButtonUp, wm_MouseMove, wm_NCHitTest, wm_NCLButtonDblClk, wm_NCLButtonDown, wm_NCLButtonUp, wm_NCMButtonDblClk, wm_NCMButtonDown, wm_NCMButtonUp, wm_NCMouseMove, wm_NCRButtonDblClk, wm_NCRButtonDown, wm_NCRButtonUp, wm_RButtonDblClk, wm_RButtonDown,* and *wm_RButtonUp*.

wm_NCLButtonDblClk

Notifies a window that the left mouse button was double-clicked in a non-client area.

Parameters *wParam*: *wParam* is one of the *ht* constants. See "(ht) Hit test codes" in Chapter 1, "Windows styles and constants". These are the same values returned by the *wm_NCHitTest* message.

lParamLo: *lParamLo* is the x-coordinate of the mouse cursor.

lParamHi: *lParamHi* is the y-coordinate of the mouse cursor.

Return value Not used.

Comments The mouse coordinates are relative to the upper left corner of the screen. A double-click is actually two single-clicks within the system's double-click time limit. A double click results in a down-click message, an up-click message, a double-click message, and a second up-click message. The default action taken by *DefWindowProc* includes sending appropriate *wm_SysCommand* messages depending on the non-client area involved. See also *wm_LButtonDblClk*, *wm_LButtonDown*, *wm_LButtonUp*, *wm_MButtonDblClk*, *wm_MButtonDown*, *wm_MButtonUp*, *wm_MouseMove*, *wm_NCHitTest*, *wm_NCLButtonDown*, *wm_NCLButtonUp*, *wm_NCMButtonDblClk*, *wm_NCMButtonDown*, *wm_NCMButtonUp*, *wm_NCMouseMove*, *wm_NCRButtonDblClk*, *wm_NCRButtonDown*, *wm_NCRButtonUp*, *wm_RButtonDblClk*, *wm_RButtonDown*, and *wm_RButtonUp*.

wm_NCLButtonDown

Notifies a window that the left mouse button was clicked in a non-client area.

Parameters *wParam*: *wParam* is one of the *ht* constants. See "(ht) Hit test codes" in Chapter 1, "Windows styles and constants". These are the same values returned by the *wm_NCHitTest* message.

lParamLo: *lParamLo* is the x-coordinate of the mouse cursor.

lParamHi: *lParamHi* is the y-coordinate of the mouse cursor.

Return value Not used.

Comments The mouse coordinates are relative to the upper left corner of the screen. See also *wm_LButtonDblClk*, *wm_LButtonDown*, *wm_LButtonUp*, *wm_MButtonDblClk*, *wm_MButtonDown*, *wm_MButtonUp*, *wm_MouseMove*, *wm_NCHitTest*, *wm_NCLButtonDblClk*, *wm_NCLButtonUp*, *wm_NCMButtonDblClk*, *wm_NCMButtonDown*, *wm_NCMButtonUp*, *wm_NCMouseMove*, *wm_NCRButtonDblClk*, *wm_NCRButtonDown*, *wm_NCRButtonUp*, *wm_RButtonDblClk*, *wm_RButtonDown*, and *wm_RButtonUp*. The default action taken by *DefWindowProc* includes sending appropriate *wm_SysCommand* messages depending on the non-client area involved.

wm_NCLButtonUp

Notifies a window that the left mouse button was released in a non-client area.

Parameters *wParam*: *wParam* is one of the *ht* constants. See "(ht) Hit test codes" in Chapter 1, "Windows styles and constants". These are the same values returned by the *wm_NCHitTest* message.

lParamLo: *lParamLo* is the x-coordinate of the mouse cursor.

lParamHi: *lParamHi* is the y-coordinate of the mouse cursor.

Return value Not used.

Comments The mouse coordinates are relative to the upper left corner of the screen. See also *wm_LButtonDblClk*, *wm_LButtonDown*, *wm_LButtonUp*, *wm_MButtonDblClk*, *wm_MButtonDown*, *wm_MButtonUp*, *wm_MouseMove*, *wm_NCHitTest*, *wm_NCLButtonDblClk*, *wm_NCLButtonDown* *wm_NCMButtonDblClk*, *wm_NCMButtonDown*, *wm_NCMButtonUp*, *wm_NCMouseMove*, *wm_NCRButtonDblClk*, *wm_NCRButtonDown*, *wm_NCRButtonUp*, *wm_RButtonDblClk*, *wm_RButtonDown*, and *wm_RButtonUp*. The default action taken by *DefWindowProc* includes sending appropriate *wm_SysCommand* messages depending on the non-client area involved.

wm_NCMButtonDblClk

Notifies a window that the middle mouse button was double-clicked in a non-client area.

Parameters *wParam*: *wParam* is one of the HT constants. See "(HT) Hit test codes" in Chapter 1, "Windows styles and constants". These are the same values returned by the *wm_NCHitTest* message.

lParamLo: *lParamLo* is the x-coordinate of the mouse cursor.

lParamHi: *lParamHi* is the y-coordinate of the mouse cursor.

Return value Not used.

Comments The mouse coordinates are relative to the upper left corner of the screen. A double-click is actually two single-clicks within the system's double-click time limit. A double click results in a down-click message, an up-click message, a double-click message, and a second up-click message. See also *wm_LButtonDblClk*, *wm_LButtonDown*, *wm_LButtonUp*,

wm_MButtonDblClk, *wm_MButtonDown*, *wm_MButtonUp*, *wm_MouseMove*, *wm_NCHitTest*, *wm_NCLButtonDblClk*, *wm_NCLButtonDown*, *wm_NCLButtonUp*, *wm_NCMButtonDown*, *wm_NCMButtonUp*, *wm_NCMouseMove*, *wm_NCRButtonDblClk*, *wm_NCRButtonDown*, *wm_NCRButtonUp*, *wm_RButtonDblClk*, *wm_RButtonDown*, and *wm_RButtonUp*. The default action taken by *DefWindowProc* includes sending appropriate *wm_SysCommand* messages depending on the non-client area involved.

wm_NCMButtonDown

Notifies a window that the middle mouse button was clicked in a non-client area.

Parameters *wParam*: *wParam* is one of the *ht* constants. See "(ht) Hit test codes" in Chapter 1, "Windows styles and constants". These are the same values returned by the *wm_NCHitTest* message.

lParamLo: *lParamLo* is the x-coordinate of the mouse cursor.

lParamHi: *lParamHi* is the y-coordinate of the mouse cursor.

Return value Not used.

Comments The mouse coordinates are relative to the upper left corner of the screen. See also *wm_LButtonDblClk*, *wm_LButtonDown*, *wm_LButtonUp*, *wm_MButtonDblClk*, *wm_MButtonDown*, *wm_MButtonUp*, *wm_MouseMove*, *wm_NCHitTest*, *wm_NCLButtonDblClk*, *wm_NCLButtonDown*, *wm_NCLButtonUp*, *wm_NCMButtonDblClk*, *wm_NCMButtonUp*, *wm_NCMouseMove*, *wm_NCRButtonDblClk*, *wm_NCRButtonDown*, *wm_NCRButtonUp*, *wm_RButtonDblClk*, *wm_RButtonDown*, and *wm_RButtonUp*. The default action taken by *DefWindowProc* includes sending appropriate *wm_SysCommand* messages depending on the non-client area involved.

wm_NCMButtonUp

Notifies a window that the middle mouse button was released in a non-client area.

Parameters *wParam*: *wParam* is one of the *ht* constants. See "(ht) Hit test codes" in Chapter 1, "Windows styles and constants". These are the same values returned by the *wm_NCHitTest* message.

lParamLo: *lParamLo* is the x-coordinate of the mouse cursor.

lParamHi: *lParamHi* is the y-coordinate of the mouse cursor.

Return value Not used.

Comments The mouse coordinates are relative to the upper left corner of the screen. See also *wm_LButtonDblClk*, *wm_LButtonDown*, *wm_LButtonUp*, *wm_MButtonDblClk*, *wm_MButtonDown*, *wm_MButtonUp*, *wm_MouseMove*, *wm_NCHitTest*, *wm_NCLButtonDblClk*, *wm_NCLButtonDown*, *wm_NCLButtonUp*, *wm_NCMButtonDblClk*, *wm_NCMButtonDown*, *wm_NCMouseMove*, *wm_NCRButtonDblClk*, *wm_NCRButtonDown*, *wm_NCRButtonUp*, *wm_RButtonDblClk*, *wm_RButtonDown*, and *wm_RButtonUp*. The default action taken by *DefWindowProc* includes sending appropriate *wm_SysCommand* messages depending on the non-client area involved.

wm_NCMouseMove

Notifies a window that the mouse has moved within a non-client area of the window.

Parameters *wParam*: *wParam* is one of the *ht* constants. See "(ht) Hit test codes" in Chapter 1, "Windows styles and constants". These are the same values returned by the *wm_NCHitTest* message.

lParamLo: *lParamLo* is the x-coordinate of the mouse cursor.

lParamHi: *lParamHi* is the y-coordinate of the mouse cursor.

Return value Not used.

Comments The mouse coordinates are relative to the upper left corner of the screen. See also *wm_LButtonDblClk*, *wm_LButtonDown*, *wm_LButtonUp*, *wm_MButtonDblClk*, *wm_MButtonDown*, *wm_MButtonUp*, *wm_MouseMove*, *wm_NCLButtonDblClk*, *wm_NCLButtonDown*, *wm_NCLButtonUp*, *wm_NCMButtonDblClk*, *wm_NCMButtonDown*, *wm_NCMButtonUp*, *wm_NCRButtonDblClk*, *wm_NCRButtonDown*, *wm_NCRButtonUp*, *wm_RButtonDblClk*, *wm_RButtonDown*, and *wm_RButtonUp*. The default action taken by *DefWindowProc* includes sending appropriate *wm_SysCommand* messages depending on the non-client area involved.

wm_NCPaint

Notifies a window that its frame needs to be painted.

Parameters *wParam*: Not used.

lParam: Not used.

Return value Not used.

Comments An application can respond to this message by painting its own custom window frame or pass it to *DefWindowProc* which paints a standard window frame. The clipping region for a custom frame will always be rectangular even if the frame is not.

wm_NCRButtonDblClk

Notifies a window that the right mouse button was double-clicked in a non-client area.

Parameters *wParam*: *wParam* is one of the *ht* constants. See "(ht) Hit test codes" in Chapter 1, "Windows styles and constants". These are the same values returned by the *wm_NCHitTest* message.

lParamLo: *lParamLo* is the x-coordinate of the mouse cursor.

lParamHi: *lParamHi* is the y-coordinate of the mouse cursor.

Return value Not used.

Comments The mouse coordinates are relative to the upper left corner of the screen. A double-click is actually two single-clicks within the system's double-click time limit. A double click results in a down-click message, an up-click message, a double-click message, and a second up-click message. See also *wm_LButtonDblClk*, *wm_LButtonDown*, *wm_LButtonUp*, *wm_MButtonDblClk*, *wm_MButtonDown*, *wm_MButtonUp*, *wm_MouseMove*, *wm_NCHitTest*, *wm_NCLButtonDblClk*, *wm_NCLButtonDown*, *wm_NCLButtonUp*, *wm_NCMButtonDblClk*, *wm_NCMButtonDown*, *wm_NCMButtonUp*, *wm_NCMouseMove*, *wm_NCRButtonDown*, *wm_NCRButtonUp*, *wm_RButtonDblClk*, *wm_RButtonDown*, and *wm_RButtonUp*. The default action taken by *DefWindowProc* includes sending appropriate *wm_SysCommand* messages depending on the non-client area involved.

wm_NCRButtonDown

Notifies a window that the right mouse button was clicked in a non-client area.

Parameters *wParam*: *wParam* is one of the *ht* constants. See "(ht) Hit test codes" in Chapter 1, "Windows styles and constants". These are the same values returned by the *wm_NCHitTest* message.

lParamLo: *lParamLo* is the x-coordinate of the mouse cursor.

lParamHi: *lParamHi* is the y-coordinate of the mouse cursor.

Return value Not used.

Comments The mouse coordinates are relative to the upper left corner of the screen. See also *wm_LButtonDblClk*, *wm_LButtonDown*, *wm_LButtonUp*, *wm_MButtonDblClk*, *wm_MButtonDown*, *wm_MButtonUp*, *wm_MouseMove*, *wm_NCHitTest*, *wm_NCLButtonDblClk*, *wm_NCLButtonDown*, *wm_NCLButtonUp*, *wm_NCMButtonDblClk*, *wm_NCMButtonDown*, *wm_NCMButtonUp*, *wm_NCMouseMove*, *wm_NCRButtonDblClk*, *wm_NCRButtonUp*, *wm_RButtonDblClk*, *wm_RButtonDown*, and *wm_RButtonUp*. The default action taken by *DefWindowProc* includes sending appropriate *wm_SysCommand* messages depending on the non-client area involved.

wm_NCRButtonUp

Notifies a window that the right mouse button was released in a non-client area.

Parameters *wParam*: *wParam* is one of the *ht* constants. See "(ht) Hit test codes" in Chapter 1, "Windows styles and constants". These are the same values returned by the *wm_NCHitTest* message.

lParamLo: *lParamLo* is the x-coordinate of the mouse cursor.

lParamHi: *lParamHi* is the y-coordinate of the mouse cursor.

Return value Not used.

Comments The mouse coordinates are relative to the upper left corner of the screen. See also *wm_LButtonDblClk*, *wm_LButtonDown*, *wm_LButtonUp*, *wm_MButtonDblClk*, *wm_MButtonDown*, *wm_MButtonUp*, *wm_MouseMove*, *wm_NCHitTest*, *wm_NCLButtonDblClk*, *wm_NCLButtonDown*, *wm_NCLButtonUp*, *wm_NCMButtonDblClk*, *wm_NCMButtonDown*,

wm_NCMButtonUp, *wm_NCMouseMove*, *wm_NCRButtonDblClk*, *wm_NCRButtonDown*, *wm_RButtonDblClk*, *wm_RButtonDown*, and *wm_RButtonUp*. The default action taken by *DefWindowProc* includes sending appropriate *wm_SysCommand* messages depending on the non-client area involved.

wm_NextDlgCtl

Changes a dialog box's control focus.

Parameters *wParam*: If *lParam* is non-zero, *wParam* is a handle to the control which is to receive the control focus, otherwise if *wParam* is zero, the next control with tab-stop style receives the focus and if *wParam* is non-zero the previous control with tab-stop style receives the focus.

lParam: If *lParam* is zero, *wParam* determines which direction to look for the next control with tab-stop style, otherwise, *wParam* is a handle to the control which is to receive the control focus.

Return value Not Used.

Comments Unlike the *SetFocus* function, this message alters the border around the default control. The *PostMessage* function should be used to send this message instead of the *SendMessage* unless your application does not concurrently handle any other messages that set the control focus.

wm_Paint

Notifies a window that it needs to repaint all or part of its client area.

Parameters *wParam*: Not used.

lParam: Not used.

Return value Not used.

Comments This message is sent when the *UpdateWindow* function is called or when the *DispatchMessage* function encounters a *wm_Paint* message. See also the *BeginPaint* and *EndPaint* functions.

wm_PaintClipboard

Asks the clipboard owner to display all or part of the clipboard contents.

Parameters *wParam*: *wParam* is a handle to the clipboard application window.

lParamLo: *lParamLo* is a handle to memory containing a *TPaintStruct* structure which defines the part of the clipboard application window's client are to be painted.

lParamHi: *lParamHi* is not used.

Return value Not used.

Comments The handle to the *TPaintStruct* data must be locked using the *GlobalLock* function before using it and unlocked using the *GlobalUnlock* function before returning from this message or yielding control. The drawing dimensions found in the *TPaintStruct* should be compared to the dimensions received in the last *wm_SizeClipboard* message. Messages which will be sent to the clipboard owner when the clipboard format is *cf_OwnerDisplay* are *wm_AskCBFormatName, wm_HScrollClipboard, wm_PaintClipboard, wm_SizeClipboard,* and *wm_VScrollClipboard*. The clipboard data and format are set by using the *SetClipboardData* function.

wm_PaintIcon

Tells a minimized window which has a class icon that it must paint its icon.

Parameters *wParam*: Not used.

lParam: Not used.

Return value Not used.

Comments The *wm_Paint* message is sent instead if there is no class icon defined for the window. The default action taken by *DefWindowProc* is to paint the window's icon with the class icon.

wm_PaletteChanged

Notifies all windows that the system palette has changed.

Parameters *wParam*: *wParam* is the handle of the window that changed the system palette.

lParam: Not used.

Return value Not used.

Comments This message is sent when the window with input focus realizes its logical palette and changes the system palette. Windows other than the one with its handle in *wParam* may realize their palettes and update their client areas in response to this message.

wm_ParentNotify

Notifies all ancestor windows of a child window when that child window is created, destroyed, or clicked over with the mouse.

Parameters *wParam*: *wParam* is *wm_Create*, *wm_Destroy*, *wm_LButtonDown*, *wm_MButtonDown*, or *wm_RButtonDown* depending on the reason this message was sent.

lParamLo: If *wParam* is *wm_Create* or *wm_Destroy*, *lParamLo* is the handle to the child window, otherwise *lParamLo* is the x-coordinate of the mouse cursor.

lParamHi: If *wParam* is *wm_Create* or *wm_Destroy*, *lParamHi* is the ID of the child window, otherwise *lParamHi* is the y-coordinate of the mouse cursor.

Return value Not used.

Comments This message is sent just before the *CreateWindow* or *CreateWindowEx* function that creates the child window returns and before any action is taken to destroy the child window. This message is sent to all ancestor windows of the child window. This message is not sent if the child window is created with the *ws_ex_NoParentNotify* extended window style. Child windows in a dialog box have the *ws_ex_NoParentNotify* extended window style unless the child window is created using the *CreateWindowEx* function with the appropriate parameters.

wm_Paste

Copies the contents of the clipboard into a window at the current cursor position.

Parameters *wParam*: Not used.

lParam: Not used.

Return value Not used.

Comments If the contents of the clipboard are not in *cf_Text* format, no action is taken.

wm_QueryDragIcon

Asks a minimized window that has no icon defined for its class whether it should use the default icon cursor to drag the icon or substitute a different one.

Parameters *wParam*: Not used.

lParam: Not used.

Return value If 0 is returned, Windows will use the default icon cursor, otherwise the low-order word of the return value is a handle of a cursor to use instead. If a cursor handle is returned, it must be a monochrome cursor compatible with the display driver's resolution. The *LoadCursor* function may be used to load a cursor from the executable resources.

wm_QueryEndSession

Asks each application wether or not the session should end.

Parameters *wParam*: Not used.

lParam: Not used.

Return value If the application can shut down, return a non-zero value, otherwise return zero.

Comments This message is sent to each application until all have returned non-zero values or one returns zero. If one application returns zero, the session will not be ended and all applications that have already been sent this message and returned a non-zero value are sent a *wm_EndSession* message with

wParam equal to zero. The default action taken by *DefWindowProc* is to return a non-zero value.

wm_QueryNewPalette

Asks a window whether or not it will realize its logical palette when it receives the input focus.

Parameters *wParam*: Not used.

lParam: Not used.

Return value If the window realizes its logical palette when it receives the input focus, returns a non-zero value, otherwise returns zero.

wm_QueryOpen

Asks an application whether it can be opened from an icon into a window.

Parameters *wParam*: Not used.

lParam: Not used.

Return value If zero is returned, the application will not be opened, otherwise the application will be opened. The default action taken in *DefWindowProc* is to return a non-zero value.

wm_Quit

Tells an application to terminate execution.

Parameters *wParam*: *wParam* is the exit code passed to Windows in the *PostQuitMessage* function call.

lParam: Not used.

Return value Not used.

Comments After this message is sent calls to the *GetMessage* function will return zero. The exit code in *wParam* must be saved and used as the exit code for the program.

wm_RButtonDblClk

Notifies a window that the right mouse button was double-clicked.

Parameters *wParam*: *wParam* is a value which indicates which virtual keys are down. It is a combination of the *mk_* constants. See "(mk_) Key state masks" in Chapter 1, "Windows styles and constants".

lParamLo: *lParamLo* is the x-coordinate of the mouse cursor.

lParamHi: *lParamHi* is the y-coordinate of the mouse cursor.

Return value Not used.

Comments This message only applies to windows created with the *cs_DblClks* style. The mouse coordinates are relative to the upper left corner of the window. A double-click is actually two single-clicks within the system's double-click time limit. A double click results in a down-click message, an up-click message, a double-click message, and a second up-click message. See also *wm_LButtonDblClk*, *wm_LButtonDown*, *wm_LButtonUp*, *wm_MButtonDblClk*, *wm_MButtonDown*, *wm_MButtonUp*, *wm_MouseMove*, *wm_NCLButtonDblClk*, *wm_NCLButtonDown*, *wm_NCLButtonUp*, *wm_NCMButtonDblClk*, *wm_NCMButtonDown*, *wm_NCMButtonUp*, *wm_NCMouseMove*, *wm_NCRButtonDblClk*, *wm_NCRButtonDown*, *wm_NCRButtonUp*, *wm_RButtonDown*, and *wm_RButtonUp*.

wm_RButtonDown

Notifies a window that the right mouse button was clicked.

Parameters *wParam*: *wParam* is a value which indicates which virtual keys are down. It is a combination of the *mk_* constants. See "(mk_) Key state masks" in Chapter 1, "Windows styles and constants".

lParamLo: *lParamLo* is the x-coordinate of the mouse cursor.

lParamHi: *lParamHi* is the y-coordinate of the mouse cursor.

Return value Not used.

Comments The mouse coordinates are relative to the upper left corner of the window. See also *wm_LButtonDblClk*, *wm_LButtonDown*, *wm_LButtonUp*, *wm_MButtonDblClk*, *wm_MButtonDown*, *wm_MButtonUp*, *wm_MouseMove*, *wm_NCLButtonDblClk*, *wm_NCLButtonDown*, *wm_NCLButtonUp*, *wm_NCMButtonDblClk*, *wm_NCMButtonDown*, *wm_NCMButtonUp*,

wm_NCMouseMove, *wm_NCRButtonDblClk*, *wm_NCRButtonDown*, *wm_NCRButtonUp*, *wm_RButtonDblClk*, and *wm_RButtonUp*.

wm_RButtonUp

Notifies a window that the right mouse button was released.

Parameters *wParam*: *wParam* is a value which indicates which virtual keys are down. It is a combination of the *mk_* constants. See "(mk_) Key state masks" in Chapter 1, "Windows styles and constants".

lParamLo: *lParamLo* is the x-coordinate of the mouse cursor.

lParamHi: *lParamHi* is the y-coordinate of the mouse cursor.

Return value Not used.

Comments The mouse coordinates are relative to the upper left corner of the window. See also *wm_LButtonDblClk*, *wm_LButtonDown*, *wm_LButtonUp*, *wm_MButtonDblClk*, *wm_MButtonDown*, *wm_MButtonUp*, *wm_MouseMove*, *wm_NCLButtonDblClk*, *wm_NCLButtonDown*, *wm_NCLButtonUp*, *wm_NCMButtonDblClk*, *wm_NCMButtonDown*, *wm_NCMButtonUp*, *wm_NCMouseMove*, *wm_NCRButtonDblClk*, *wm_NCRButtonDown*, *wm_NCRButtonUp*, *wm_RButtonDblClk*, and *wm_RButtonDown*.

wm_RenderAllFormats

Asks the clipboard owner to render the clipboard data in all formats it knows.

Parameters *wParam*: Not used.

lParam: Not used.

Return value Not used.

Comments This message is sent to the application of the clipboard owner when the application is to be destroyed. Each format should be formatted and the handle to the formatted data should be sent to the clipboard using the *SetClipboardData* function. See also *wm_RenderFormat*.

wm_RenderFormat

Asks the clipboard owner to send a handle to data in the requested format to the clipboard.

Parameters *wParam*: *wParam* is the requested data format for the *SetClipboardData* function.

lParam: Not used.

Return value Not used.

Comments The data should be formatted as requested. The handle to the formatted data must be sent to the clipboard using the *SetClipboardData* function. See also *wm_RenderAllFormats*.

wm_SetCursor

Notifies a parent window that the cursor has moved.

Parameters *wParam*: *wParam* is a handle to the window that contains the cursor.

lParamLo: *lParamLo* is one of the *ht* constants. See "(ht) Hit test codes" in Chapter 1, "Windows styles and constants". These are the same values returned by the *wm_NCHitTest* message.

lParamHi: *lParamHi* is the mouse message number.

Return value If zero is returned, *DefWindowProc* continues with its default action, otherwise no further action is taken.

Comments The default action taken by *DefWindowProc* is to set the cursor to an arrow when in a non-client area or to the registered-class cursor when in the client area. This message allows a parent window to alter the cursor depending on which child window or area of the client area the cursor is over. This message is not sent if mouse input is captured using the *SetCapture* function. The message must be passed on to *DefWindowProc* with the original or altered parameters. If *DefWindowProc* is given this message and *lParamLo* is *htError* and *lParamHi* is a mouse button-down message number, the *MessageBeep* function is called. *lParamHi* is zero when the window enters menu mode.

wm_SetFocus

Notifies a window that it has received the input focus.

Parameters *wParam*: *wParam* is a handle to the window that just lost the input focus.

lParam: Not used.

Return value Not used.

Comments Appropriate caret functions for displaying a caret should be called at this time, if a caret is to be shown.

wm_SetFont

Sets the font used by a dialog box or notifies a dialog box that its controls are about to be created and fonts may be selected for them.

Parameters *wParam*: *wParam* is a handle to the font to use or zero if the control is to use the system font.

lParam: If *lParam* is 0, the control will not be redrawn, otherwise the control will be redrawn.

Return value Not used.

Comments A font must be deleted using the *DeleteObject* function when the font is no longer needed, for instance after the dialog box is destroyed. The size of the dialog box should be changed before the font is changed. This message is sent to dialog boxes with the *ds_SetFont* style before the dialog box's controls are created. Another good time to send this message is when the *wm_InitDialog* message is received.

wm_SetRedraw

Sets or clears a window's redraw flag.

Parameters *wParam*: *wParam* is the new redraw flag. If *wParam* is zero, redrawing is disabled, otherwise redrawing is enabled.

lParam: Not used.

Return value Not used.

wm_SetText

Sets a window's associated text.

Parameters *wParam*: Not used.

lParam: *lParam* is a pointer to a null-terminated string that is to be the associated text.

Return value If there is not enough space available to set the associated text, *lb_ErrSpace* or *cb_ErrSpace* will be returned for list boxes or combo boxes respectively. If the window is a list box and no item is selected, *lb_Err* is returned. If the window is a combo box which has no edit control, *cb_Err* is returned.

Comments For edit controls, the text is the contents of the edit control. For button controls, the text is the button name. For list boxes, the text is the currently selected item, if any. For combo boxes, the text is the contents of the combo box's edit control. For all other windows, the text is the window caption. The current selection in a combo box is not changed by this message, only the contents of the combo box's edit control. *cb_SelectString* should be used to select an entry in the combo box's list box which matches the text in the combo box's edit control. See also *wm_GetText*.

wm_ShowWindow

Notifies a window that it is about to be shown or hidden.

Parameters *wParam*: If *wParam* is zero, the window is being hidden, otherwise the window is being shown.

lParam: *lParam* is zero if this message is sent because of a *ShowWindow* function call. *lParam* is *sw_ParentClosing* if the window's parent is closing or a pop-up window is being hidden. *lParam* is *sw_ParentOpening* if the window's parent is opening or a pop-up window is being shown.

Return value Not used.

Comments 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 or opened. All pop-up windows associated with an overlapped window are hidden when the overlapped window is closed. The default action taken by *DefWindowProc* is to show or hide the window as specified.

wm_Size

Notifies a window that its size has changed.

Parameters *wParam*: *wParam* is one of the *size* constants. See "(size) Size constants" in Chapter 1, "Windows styles and constants".

lParamLo: *lParamLo* is the new width of the client area of the window.

lParamHi: *lParamHi* is the new height of the client area of the window.

Return value Not used.

Comments If the *SetScrollPos* or *MoveWindow* function is used when handling this message, the *Redraw* parameter for *SetScrollPos* or the *Repaint* parameter for *MoveWindow* should be nonzero so the window will be repainted.

wm_SizeClipboard

Notifies the clipboard owner that the clipboard application window has changed size.

Parameters *wParam*: *wParam* is a handle to the clipboard application window.

lParamLo: *lParamLo* is a *TRect* structure specifying the area of the clipboard application window to paint.

lParamHi: Not used.

Return value Not used.

Comments If *wParam* is zero (the *TRect* is (0,0,0,0)) the clipboard application is about to be destroyed or minimized and the clipboard owner may free any display resources. The *TPaintStruct* structure used to paint the clipboard application window must be locked in place using the *GlobalLock* function and unlocked using the *GlobalUnlock* function before returning from this message. The *TRect* in *lParamLo* should be copied for use by the next *wm_PaintClipboard* message. Messages which will be sent to the clipboard owner when the clipboard format is *cf_OwnerDisplay* are *wm_AskCBFormatName*, *wm_HScrollClipboard*, *wm_PaintClipboard*, *wm_SizeClipboard*, and *wm_VScrollClipboard*. The format of the clipboard is set when the clipboard data is set via the *SetClipboardData* function.

wm_SpoolerStatus

Notifies an application that a job has been added to or removed from the Print Manager queue.

Parameters *wParam*: *wParam* is set to *pr_JobStatus*.

lParamLo: *lParamLo* is the number of jobs currently in the Print Manager queue.

lParamHi: Not used.

Return value Not used.

Comments This message is informational only. It is sent by the Print Manager application.

wm_SysChar

Notifies the window with the focus that a system key was pressed or notifies the active window that a key was pressed when no window has the input focus.

Parameters *wParam*: *wParam* is the key value.

lParamLo: *lParamLo* is the number of times this key-press was repeated because the key is being held down.

lParamHi: Bits 0-7 of *lParamHi* is the key's scan code which is OEM-dependent. Bit 8 is 1 if the key is an extended key. Bit 13 is 1 if the *Alt* key was down while the key was pressed. If the key was down before this message was sent, bit 14 is 1. Bit 15 is 1 if the key is being released, zero if it is being pressed.

Return value Not used.

Comments For this message bit 15 of *lParamHi* will be zero. Bit 13 of *lParam* will be 1 if a system key was pressed or zero if no window has the input focus. If bit 13 of *lParam* is zero, this message may be passed to the *TranslateAccelerator* function so that accelerators can be used in the active window even if it does not have the input focus. If no window has the focus, *wm_SysKeyDown*, *wm_SysChar*, and *wm_SysKeyUp* messages are sent instead of *wm_KeyDown*, *wm_Char*, and *wm_KeyUp* messages. See also *wm_Char*, *wm_DeadChar*, and *wm_SysDeadChar*.

wm_SysColorChange

Notifies a top-level window that the system color settings have changed.

Parameters *wParam*: Not used.

lParam: Not used.

Return value Not used.

Comments A *wm_Paint* message is sent to any window affected by a system color change. Applications should delete any brushes that use existing system colors and recreate them using the new system colors.

wm_SysCommand

Notifies a window that a System menu item, the minimize box, or the maximize box has been selected.

Parameters *wParam*: *wParam* is the system command request. It can be any of the *sc_* constants. See "(sc_) System command values" in Chapter 1, "Windows styles and constants". The bottom 4 bits of *wParam* are used internally by Windows. They should be set to 0 before *wParam* is used.

lParamLo: *lParamLo* is the x-coordinate of the mouse or zero if the mouse was not used.

lParamHi: *lParamHi* is the y-coordinate of the mouse or zero if the mouse was not used.

Return value Not used.

Comments Accelerator keys that map to System-menu items, translate into *wm_SysCommand* message instead of *wm_Command* messages. *wm_Command* is sent for accelerators only if the window is not minimized or the window is minimized and the accelerator does not match any menu items on the window's menu or the System-menu. System-menu items may be modified using the *GetSystemMenu, AppendMenu, InsertMenu,* and *ModifyMenu* functions. An application must handle all modified System-menu entry selections. All messages an application does not handle must be passed to *DefWindowProc*. *wm_SysCommand* messages may be sent to *DefWindowProc* any time an application needs to carry out a System-menu command.

wm_SysDeadChar

Notifies a window of a dead system character.

Parameters *wParam*: *wParam* is the key value.

lParamLo: *lParamLo* is the number of times this key-press has been repeated.

lParamHi: *lParamHi* is the number of times this key-press has been repeated because the key is being held down.

Return value Not used.

Comments Dead keys include the umlaut character and accents. A dead system key is a dead key in combination with the *Alt* key. This message can be used to give feedback for keys pressed that do not necessarily result in a character by themselves. See also *wm_Char*, *wm_DeadChar*, and *wm_SysChar*.

wm_SysKeyDown

Notifies the window with the focus that a system key was pressed or notifies the active window that a key was pressed when no window has the input focus.

Parameters *wParam*: *wParam* is the virtual-key code.

lParamLo: *lParamLo* is the number of times this key-press was repeated because the key is being held down.

lParamHi: Bits 0-7 of *lParamHi* is the key's scan code which is OEM-dependent. Bit 8 is 1 if the key is an extended key. Bit 13 is 1 if the *Alt* key was down while the key was pressed. If the key was down before this message was sent, bit 14 is 1. Bit 15 is 1 if the key is being released, zero if it is being pressed.

Return value Not used.

Comments For this message bit 15 of *lParamHi* will be zero. Bit 13 of *lParam* will be 1 if a system key was pressed or zero if no window has the input focus. If bit 13 of *lParam* is zero, this message may be passed to the *TranslateAccelerator* function so that accelerators can be used in the active window even if it does not have the input focus. Due to auto-repeat, more than one *wm_SysKeyDown* message may be sent before a *wm_SysKeyUp* message is sent. If no window has the focus, *wm_SysKeyDown*,

wm_SysChar, and *wm_SysKeyUp* messages are sent instead of *wm_KeyDown*, *wm_Char*, and *wm_KeyUp* messages.

wm_SysKeyUp

Notifies the window with the focus that a system key was released or notifies the active window that a key was released when no window has the input focus.

Parameters *wParam*: *wParam* is the virtual-key code.

lParamLo: *lParamLo* is the number of times this key-press was repeated because the key was held down.

lParamHi: Bits 0-7 of *lParamHi* is the key's scan code which is OEM-dependent. Bit 8 is 1 if the key is an extended key. Bit 13 is 1 if the *Alt* key was down while the key was pressed. If the key was down before this message was sent, bit 14 is 1. Bit 15 is 1 if the key is being released, zero if it is being pressed.

Return value Not used.

Comments For this message bit 15 of *lParamHi* will be 1. Bit 13 of *lParam* will be 1 if a system key was released or zero if no window has the input focus. If bit 13 of *lParam* is zero, this message may be passed to the *TranslateAccelerator* function so that accelerators can be used in the active window even if it does not have the input focus. Due to auto-repeat, more than one *wm_SysKeyDown* message may be sent before a *wm_SysKeyUp* message is sent. If no window has the focus, *wm_SysKeyDown*, *wm_SysChar*, and *wm_SysKeyUp* messages are sent instead of *wm_KeyDown*, *wm_Char*, and *wm_KeyUp* messages.

wm_TimeChange

Notifies a top-level window that the system time has been changed.

Parameters *wParam*: Not used.

lParam: Not used.

Return value Not used.

Comments After an application makes changes to the system time, it must send this message to all top-level windows by using the *SendMessage* function.

wm_Timer

Notifies an application that a timer's time limit is elapsed.

Parameters *wParam*: *wParam* is the timer ID.

lParam: *lParam* is used only when the timer function passed to the *SetTimer* function when the timer was created was not zero. In that case the message is not put in the message queue, but passed directly to the timer function.

Return value Not used.

wm_Undo

Undoes the last operation on an edit control.

Parameters *wParam*: Not used.

lParam: Not used.

Return value Not used.

wm_VKeyToItem

Asks the owner of a list box what the list box should do in response to a *wm_KeyDown* message.

Parameters *wParam*: *wParam* is the key pressed.

lParamLo: *lParamLo* is a handle to the list box.

lParamHi: *lParamHi* is the current caret position.

Return value If –2 is returned, the application handled everything. If –1 is returned, the list box is to perform the default action for that key. If zero or greater is returned, the list box is to perform the default action for that key, but on the item specified by the return value. This message only applies to list boxes with the *lbs_WantKeyboardInput* style. See also *wm_CharToItem*.

wm_VScroll

Notifies a window that a vertical scroll bar has been clicked.

Parameters *wParam*: *wParam* is a scroll bar code which describes the effect of the scroll bar click. It may be any of the *sb_* constants that apply to vertical scroll bar controls. See "(sb_) Scroll bar commands" in Chapter 1, "Windows styles and constants".

lParamLo: Not used.

lParamHi: *lParamHi* is the handle of the scroll bar control. *lParamHi* will be zero if the scroll bar control is a control created with the window by using the *ws_VScroll* style.

Return value Not used.

Comments If an application scrolls the text in a window, it must also use the *SetScrollPos* function to reset the scroll bar's thumb position.

wm_VScrollClipboard

Notifies the owner of a clipboard with the *cf_OwnerDisplay* format that the vertical scroll bar in the clipboard application has been clicked.

Parameters *wParam*: *wParam* is a handle to the clipboard application window.

lParamLo: *lParamLo* is a scroll bar code which describes the effect of the scroll bar click. It may be any of the *sb_* constants that apply to vertical scroll bar controls. See "(sb_) Scroll bar commands" in Chapter 1, "Windows styles and constants".

lParamHi: Not used.

Return value Not used.

Comments The clipboard owner must repaint the clipboard application window or use the *InvalidateRect* function. The clipboard application window scroll bar position must be reset using the *SetScrollPos* function. Messages which will be sent to the clipboard owner when the clipboard format is *cf_OwnerDisplay* are *wm_AskCBFormatName*, *wm_HScrollClipboard*, *wm_PaintClipboard*, *wm_SizeClipboard*, and *wm_VScrollClipboard*. The clipboard data and format are set by using the *SetClipboardData* function.

wm_WinIniChange

Notifies a top-level window that the Windows initialization file, WIN.INI, has changed.

Parameters *wParam*: Not used.

lParam: Points to a section name string.

Return value Not used.

Comments Any time an application changes the Windows initialization file it should send this message to all top-level windows using the *SendMessage* function. *lParam* must not be **nil**.

C H A P T E R

4

Windows type reference

Windows defines a number of types and record structures. ObjectWindows provides Turbo Pascal equivalents for them, defined in the *WinTypes* unit. Each of them is documented here.

Bool type — WinTypes

Declaration `Bool = System.WordBool;`

Bool is exactly equivalent to the standard Turbo Pascal type *WordBool*. It is provided for compatibility with Windows code written in other languages.

HBitmap type — WinTypes

Declaration `HBitmap = THandle;`

HBitmap is a handle type for bitmap handles.

HBrush type — WinTypes

Declaration `HBrush = THandle;`

HBrush defines a handle type for brush drawing tools.

HCursor type — WinTypes

Declaration `HCursor = THandle;`

HCursor defines a handle type for cursor handles.

HDC type — WinTypes

Declaration `HDC = THandle;`

HDC defines a handle type for device context handles. Display contexts are a kind of device context, so display context handles are always stored in variables of type *HDC*.

HFont type — WinTypes

Declaration `HFont = THandle;`

HFont defines a handle type for font drawing tools.

HIcon type — WinTypes

Declaration `HIcon = THandle;`

HIcon defines a handle type for icon handles.

HMenu type — WinTypes

Declaration `HMenu = THandle;`

HMenu defines a handle type for menu resources.

HPalette type — WinTypes

Declaration `HPalette = THandle;`

HPalette defines a handle type for palette handles.

HPen type — WinTypes

Declaration `HPen = THandle;`

HPen defines a handle type for pen drawing tools.

HRgn type — WinTypes

Declaration `HRgn = THandle;`

HRgn defines a handle type for region handles.

HStr type — WinTypes

Declaration `HStr = THandle;`

HStr defines a handle type for string handles.

HWnd type — WinTypes

Declaration `HWnd = THandle;`

HWnd defines a handle type for window handles. These are commonly used by ObjectWindows interface objects to keep track of their associated Windows interface elements. Many API function calls require a window handle to indicate which window they are to act on.

LPHandle type — WinTypes

Declaration `LPHandle = PHandle;`

LPHandle defines a long pointer to a handle. It is not used by ObjectWindows, but is included for compatibility with Windows code written in other languages.

LPVoid type — WinTypes

Declaration `LPVoid = Pointer;`

LPVoid defines a long pointer. It is not used by ObjectWindows, but is included for compatibility with Windows code written in other languages.

MakeIntAtom type — WinTypes

Declaration `MakeIntAtom = PStr;`

MakeIntAtom is used for typecasting integer numbers into atoms. It is equivalent to typecasting into the Turbo Pascal type *PChar*.

MakeIntResource type — WinTypes

Declaration `MakeIntResource = PStr;`

MakeIntResource is used for typecasting integer numbers into resource names. It is equivalent to typecasting into the Turbo Pascal type *PChar*.

PBool type — WinTypes

Declaration `PBool = ^WordBool;`

PBool defines a pointer to a 16-bit Boolean value.

PByte type — WinTypes

Declaration `PByte = ^Byte;`

PByte defines a pointer to an 8-bit unsigned value.

PHandle type — WinTypes

Declaration `PHandle = ^THandle;`

PHandle defines a pointer to a generic Windows handle.

PInteger type — WinTypes

Declaration `PInteger = ^Integer;`

PInteger defines a pointer to a signed, 16-bit integer number.

PLongint type — WinTypes

Declaration

```
PLongint = ^Longint;
```

PLongint defines a pointer to a 32-bit integer number.

PStr type — WinTypes

Declaration

```
PStr = PChar;
```

PStr defines a pointer to a null-terminated string. It is exactly equivalent to the Turbo Pascal type *PChar*, and is provided for compatibility with code written in other languages.

PWord type — WinTypes

Declaration

```
PWord = ^Word;
```

PWord defines a pointer to an unsigned 16-bit integer number.

TAtom type — WinTypes

Declaration

```
TAtom = Word;
```

TAtom defines a 16-bit number identifying an atom, or message, to be sent between DDE-compliant applications.

TBitmap type — WinTypes

Declaration

```
TBitmap = record
  bmType: Integer;
  bmWidth: Integer;
  bmHeight: Integer;
  bmWidthBytes: Integer;
  bmPlanes: Byte;
  bmBitsPixel: Byte;
  bmBits: Pointer;
end;
```

The *TBitmap* record is used by the *CreateBitmapIndirect* and *GetObject* functions to describe the size, colors, and bit values for a bitmap.

The *bmType* field defines the bitmap type. A zero value indicates a logical bitmap. *bmWidth* and *bmHeight* define the width in pixels and the height in raster lines, respectively, of the bitmap. Both must be greater than zero. *bmWidthBytes* gives the number of bytes in each raster line, and must be an even number.

bmPlanes and *bmBitsPixel* give the number of color planes and the number of adjacent color bits on each plane, respectively.

bmBits is a pointer to the actual bits that make up the bitmap. The bits take the form of an array of bytes.

TBitmapCoreHeader type — WinTypes

Declaration

```
TBitmapCoreHeader = record
  bcSize: Longint;                    { used to get to color table }
  bcWidth: Word;
  bcHeight: Word;
  bcPlanes: Word;
  bcBitCount: Word;
end;
```

TBitmapCoreHeader defines the size and colors of a device-independent bitmap. *TBitmapCoreHeader* records are used as part of *TBitmapCoreInfo* records to fully define device-independent bitmaps.

bcSize is the number of bytes in the *TBitmapCoreHeader* record.

bcWidth and *bcHeight* are the width and height (in pixels) of the bitmap, respectively. *bcPlanes* gives the number of planes for the target device; it must be set to 1. *bcBitCount* gives the number of bits per pixel. Allowed values for *bcBitCount* are 1, 4, 8, and 24.

The significance of the *bcBitCount* bits is as follows:

- If *bcBitCount* is 1, the bitmap is monochrome, the color table must contain two entries, and each bit in the bitmap represents one pixel. A bit that is clear represents the first color in the table; a bit that is set represents the second color.
- If *bcBitCount* is 4, the bitmap has up to 16 colors, numbered 0 to 15, so each pixel in the bitmap requires four bits to indicate its color. The color

table contains 16 entries. Each byte in the bitmap thus represents two pixels, the high-order half-byte first, then the low-order half-byte.

- If *bcBitCount* is 8, the bitmap has up to 256 colors, so it takes a full byte to represent each pixel. Each byte in the bitmap, then, represents an index between 0 and 255 into the color table.
- If *bcBitCount* is 24, the bitmap has up to 2^{24} colors. There is no color table, and each pixel is represented by a trio of bytes, signifying the intensities of the red, green, and blue in the pixel.

TBitmapCoreInfo type — WinTypes

Declaration

```
TBitmapCoreInfo = record
  bmciHeader: TBitmapCoreHeader;
  bmciColors: array[0..0] of TRGBTriple;
end;
```

TBitmapCoreInfo records combine size and color information from a *TBitmapCoreHeader* record with a table of color information to fully define a device-independent bitmap.

bmciHeader is a *TBitmapCoreHeader* record, defining the size and color information for the bitmap. The *bcSize* field of *bmciHeader* can be used to provide an offset to the *bmciColors* field.

bmciColors is an array of *TRGBTriple* records; The number of elements in that array is determined by the *bcBitCount* field of the *bmciHeader* field. *bmciColors* can be either an array of RGB color records, or it may be an array of indexes into the currently realized logical palette. The way the field is interpreted depends on the *Usage* parameter passed to the function accessing the device-independent bitmap. See *"DIB_* Color table identifiers" in Chapter 1 for more information.

Use of RGB values for the *bmciColors* field is strongly encouraged, unless the bitmap is used exclusively by a single application. If the bitmap is stored in a file or transferred to another program, it should not use palette indexing.

TBitmapFileHeader type — WinTypes

Declaration

```
TBitmapFileHeader = record
  bfType: Word;
  bfSize: Longint;
  bfReserved1: Word;
  bfReserved2: Word;
  bfOffBits: Longint;
end;
```

The *TBitmapFileHeader* record defines the header of a device-independent bitmap file, which contains data defining the type size, and layout of the bitmap file.

A device-independent bitmap file consists of a *TBitmapFileHeader*, followed by either a *TBitmapInfo* record or a *TBitmapCoreInfo* record, then the actual bitmap data.

The fields in the bitmap file header are defined as follows.

The *bfType* field gives the type of the file, which must be BM.

The *bfSize* field gives the size of the file in 4-byte blocks.

bfReserved1 and *bfReserved2* are, as their names imply, reserved by Windows.

The *bfOffBits* field tells how many bytes into the file the actual bitmap information begins.

TBitmapInfo type — WinTypes

Declaration

```
TBitmapInfo = record
  bmiHeader: TBitmapInfoHeader;
  bmiColors: array[0..0] of TRGBQuad;
end;
```

TBitmapInfo records hold size and color information for device-independent bitmaps for Windows 3.0. The actual bitmap is defined as an array of bytes representing the pixels of the bitmap.

The *bmiHeader* field holds a *TBitmapInfoHeader* record that defines the size and color formatting for the bitmap. The *bmiColors* field is an array of

TRGBQuad records, defining the colors of the bitmap. The number of entries in the array is determined by the value of the *biBitCount* field of the *bmiHeader* record.

TBitmapInfoHeader type — WinTypes

Declaration

```
TBitmapInfoHeader = record
  biSize: Longint;
  biWidth: Longint;
  biHeight: Longint;
  biPlanes: Word;
  biBitCount: Word;
  biCompression: Longint;
  biSizeImage: Longint;
  biXPelsPerMeter: Longint;
  biYPelsPerMeter: Longint;
  biClrUsed: Longint;
  biClrImportant: Longint;
end;
```

TBitmapInforHeader records are used by *TBitmapInfo* records to define the dimensions and color formatting of a device-independent bitmap for Windows 3.0.

The *biSize* field gives the size, in bytes, of the record. *biWidth* and *biHeight* are the width and height in pixels, respectively, of the bitmap. *biPlanes* is the number of planes for the target device; It must be set to one. *biBitCount* gives the number of bits required to describe each pixel in the bitmap. *biCompression* gives the type of compression used for the bitmap; it may be any of the *bi_* constants defined in Chapter 1. *biSizeImage* is the size of the bitmap image, in bytes. *biXPelsPerMeter* and *biYPelsPerMeter* give the horizontal and vertical resolution, respectively, of the target device for the bitmap.

The *biClrUsed* field is used to specify the number of color table entries actually used by the bitmap. The *biBitCount* value determines the maximum number of entries; a zero in *biClrUsed* indicates that the maximum number is used. A *biClrUsed* between 1 and 23 indicates the actual number of colors accessed. If *biBitCount* is 24, *biClrUsed* is the size of the reference color table used by Windows to optimize color palette performance.

The *biClrImportant* field gives the number of colors deemed important for the display of the bitmap. A zero value indicates that all colors are important.

The meanings of the values of *biBitCount* correspond exactly to those defined for the *bcBitCount* field of *TBitmapCoreHeader* records.

TCatchBuf type — WinTypes

Declaration

```
TCatchBuf = array[0..8] of Integer;
```

TCatchBuf is a record that holds the state of all system registers. It is used by the *Catch* and *Throw* functions.

TClientCreateStruct type — WinTypes

Declaration

```
TClientCreateStruct = record
  hWindowMenu: THandle;
  idFirstChild: Word;
end;
```

TClientCreateStruct is used to hold window ID and menu information when creating MDI client windows. The *hWindowMenu* field is a handle to the application's menu. *idFirstChild* is the child window ID of the first child window of the MDI application. Child window IDs are uniquely assigned and maintained by Windows.

TColorRef type — WinTypes

Declaration

```
TColorRef = Longint;
```

TColorRef is a 32-bit value corresponding to a color, used by numerous GDI functions. It can be interpreted in three different ways, depending on the value of the high-order byte of the high-order word in the long integer.

If that highest-order byte is zero, the next three bytes represent RGB color intensities for blue, green, and red, respectively, so the value $00FF0000 represents full-intensity, pure blue, $0000FF00 is pure green, and $000000FF is pure red. $00000000 is black, and $00FFFFFF is white. RGB

values can most easily be converted into *TColorRef* values using the *RGB* function.

If the highest-order byte is one, the next byte must be zero. The low-order word (the next two bytes) form an index into a logical palette. Thus, $01000000 is index zero (the first entry) of a palette. Integer-type palette indexes can be converted into *TColorRef* values using the *PaletteIndex* function.

If the highest-order byte is two, the next three bytes represent RGB color intensities (as with RGB values described for a zero highest-byte), but the value will be matched to the nearest color in the logical palette in the current device context. Palette-relative RGB *TColorRef* values can be produced from RGB values with the *PaletteRGB* function.

In order for palette index or palette-relative *TColorRef* values to work with a device context, an application with its own palette must select it palette into the device context being written to (using *SelectPalette*) and realize it (using *RealizePalette*) in order to have drawing functions use the correct colors from the palette. Similarly, before creating a logical drawing tool, the palette must be selected and realized if the colors are to take effect.

TCompareItemStruct type — WinTypes

Declaration

```
TCompareItemStruct = record
  CtlType: Word;
  CtlID: Word;
  hwndItem: HWnd;
  itemID1: Word;
  itemData1: Longint;
  itemID2: Word;
  itemData2: Longint;
end;
```

The *TCompareItemStruct* record is used for comparing items in sorted owner-draw combo boxes or list boxes. Adding items to such controls produces a *wm_CompareItem* message, one of the parameters of which is a pointer to a *TCompareItemStruct*. Upon receiving the message, the owner compares the items in the record and returns values depending on the result. See "wm_CompareItem message" in Chapter 2.

The *CtlType* field holds an *odt_* constant indicating that the control is an owner-draw combo box (*odt_ComboBox*) or list box (*odt_ListBox*). *CtlID* and

hwndItem are the control ID and window handle for the control, respectively.

itemID1 and *itemData1* give the index of the first item in the listbox or combo box being compared and the data for that item. The *itemData1* information is the data from the *lParam* parameter of the message that added the item to the list. *itemID2* and *itemData2* provide the same information for the second item.

TComStat type — WinTypes

Declaration

```
TComStat = record
  Flags: Byte;
  cbInQue: Word;
  cbOutQue: Word;
end;
```

TComStat records hold status information on a communications device. They are used by the *GetCommError* function.

The *Flags* field is a bitmapped field, with the bits defined by the *com_* constants.

The *cbInQue* and *cbOutQue* fields give the number of characters in the receive and transmit queues, respectively.

TCreateStruct type — WinTypes

Declaration

```
TCreateStruct = record
  lpCreateParams: PChar;
  hInstance: THandle;
  hMenu: THandle;
  hwndParent: HWnd;
  cy: Integer;
  cx: Integer;
  y: Integer;
  x: Integer;
  style: LongInt;
  lpszName: PChar;
  lpszClass: PChar;
  dwExStyle: Longint;
end;
```

The *TCreateStruct* record is used to pass initialization parameters to the window function of an application.

The *lpCreateParams* field points to window creation data. *hInstance*, *hMenu*, and *hwndParent* are handles to the module instance of the module that owns the window, the menu for the window, and the parent window of the new window. *hwndParent* is zero if the window being created is the main window of the application.

cy and *cx* are the height and width of the window, and *y* and *x* give the vertical and horizontal coordinates of the upper left corner of the window, relative to its parent, if any.

style hold the style flags for the window, and *lpszName* and *lpszClass* point to the null-terminated strings specifying the name and class name, respectively, of the window. *ExStyle* holds extended style information for the window.

TDCB type — WinTypes

Declaration

```
TDCB = record
  Id: Byte;
  BaudRate: Word;
  ByteSize: Byte;
  Parity: Byte;
  StopBits: Byte;
  RlsTimeout: Word;
  CtsTimeout: Word;
  DsrTimeout: Word;
  Flags: Word;
  XonChar: Char;
  XoffChar: Char;
  XonLim: Word;
  XoffLim: Word;
  PeChar: Char;
  EofChar: Char;
  EvtChar: Char;
  TxDelay: Word;
end;
```

TDCB records hold control information for serial communications devices, used by the *BuildCommDCB*, *GetCommmState*, and *SetCommState* functions.

The *Id* field is the ID of the communications device. If the high-order bit is set (compare with the mask *LPTx*), the device is a parallel device. Otherwise it is a serial port.

BaudRate, *ByteSize*, *Parity*, and *StopBits* define the communications parameters for the port. *ByteSize* is the number of bits in each character, ranging from 4 to 8. Parity is one of the communications constants *EvenParity*, *MarkParity*, *NoParity*, *OddParity*, or *SpaceParity*. *StopBits* is one of the communications constants *OneStopBit*, *One5StopBits*, or *TwoStopBits*. The communications constants are defined in Chapter 1.

RlsTimeout, *CtsTimeout*, and *DsrTimeout* give the time in milliseconds the device should wait before timing out waiting for the RLSD, CTS, and DSR signals, respectively.

fBinary designates whether the port is using binary mode. In binary mode, all data flows through unchanged. In non-binary mode, the Eof character (*EofChar*) is used to signify the end of data.

In the *Flags* field, each bit represents a toggle switch on a different kind of error checking. The bits are defined in the following table. Individual bits can be accessed using the *dcb_* constants.

Table 4.1
TDCB bit flags

Bit	Meaning if set
fRtsDisable	RTS disabled
fParity	Parity-checking enabled
fOutxCtsFlow	CTS monitored on transmission
fOutxDsrFlow	DSR monitored on transmission
fDummy	Reserved
fDtrDisable	DTR disabled
fOutX	Xon/Xoff control used on transmission
fInX	Xon/Xoff control used on receive
fPeChar	Parity errors replaced
fNull	Null characters discarded
fChEvt	*EvtChar* character is flagged as an event
fDtrFlow	DTR used for receive flow control
fRtsFlow	RTS used for receive flow control
fdummy2	Reserved

XonChar and *XoffChar* give the values of the Xon and Xoff characters for both transmission and reception, respectively. *XonLim* is the number of characters in the receive queue that trigger the sending of an Xon. *XoffLim* is the number of characters in the receive queue that triggers the sending of an Xoff.

PeChar, *EofChar*, and *EvtChar* define the characters used to replace parity errors, to signal the end of data, and to signal an event, respectively.

The *TxDelay* field is currently not used.

TDDEAck type — WinTypes

Declaration

```
TDDEAck = record
  Flags: Word;
end;
```

The *TDDEAck* record holds acknowledgement information, sent in a parameter of a *wm_dde_Ack* message in response to any DDE message other than *wm_dde_Initiate*.

The *Flags* field is a bitmapped word, with only two bits currently defined for application use. The *fAck* bit, if set, indicates that the request was accepted. The *fBusy* bit, if set, indicates that the application is unable to respond to the request. *fBusy* is only meaningful if *fAck* is zero. The low-order byte of *Flags* is the *bAppReturnCode* "field," which can be accessed using the *dde_AppReturnCode* mask. It contains application-specific return codes.

The remaining bits are reserved for Windows.

TDDEAdvise type — WinTypes

Declaration

```
TDDEAdvise = record
  Flags: Word;
  cfFormat: Integer;
end;
```

The *TDDEAdvise* record holds a request to a DDE server, and is passed in a parameter of a *wm_dde_Advise* message.

The *Flags* field is a bitmapped field, with only two bits, *fAckReq* and *fDeferUpd*, defined. They can be accessed with *dde_* constants. The 14 lowest-order bits of *Flags* are undefined, but reserved.

The *fAckReq* bit, if set, signifies a request to the server to send its *wm_dde_Data* messages with their *fAckReq* bit set, as a means of implementing message-flow control. The *fDeferUpd* bit, if set, requests that the server send *wm_dde_Data* messages with their *hData* handles set to zero, to alert the client of changed data. When so alerted, the client can respond by sending a *wm_dde_Request* message to ask for the updated data.

The *cfFormat* field specifies the client's preferred data format, using one of the *cf_* Clipboard format constants.

TDDEData type — WinTypes

Declaration

```
TDDEData = record
  Flags: Word;
  cfFormat: Integer;
  Value: array[0..1] of Char;
end;
```

The *TDDEData* record holds data being transferred from one application to another. It is passed as a parameter in *wm_dde_Data* messages.

The *Flags* field is a bitmapped field, with only three bits currently defined. These bits, *fAckReq*, *fRelease*, and *fRequested*, can be accessed with *dde_* constants.

The *fAckReq* bit, if set, indicates that the client application should send an acknowledgement upon receiving the data, in the form of a *wm_dde_Ack* message. The *fRelease* bit, if set, means the client should free the data sent with the *wm_dde_Data* message after processing it. The *fRequested* bit, if set, indicates that the data attached come as a response to a request from the client. All other *Flags* bits are reserved.

The *cfFormat* field holds a *cf_* Clipboard format constant, indicating the format in which the data is sent to the client.

The *Value* field holds the data being transferred, in the format given by *cfFormat*.

TDDEPoke type — WinTypes

Declaration

```
TDDEPoke = record
  Flags: Word;
  cfFormat: Byte;
  Value: array[0..1] of Byte;
end;
```

The *TDDEPoke* record holds unsolicited data, accompanying a *wm_dde_Poke* message.

The *Flags* field is a bitmapped field, with only one bit, *fRelease*, currently defined. The *fRelease* bit indicates, if set, that the recipient should free the

data after processing it. All other *Flags* bits are reserve. *FRelease* can be accessed with the *dde_Release* constant.

The *cfFormat* field specifies the client's preferred data format, using one of the *cf_* Clipboard format constants.

The *Value* field holds the data being transferred, in the format given by *cfFormat*.

TDeleteItemStruct type — WinTypes

Declaration

```
TDeleteItemStruct = record
  CtlType: Word;
  CtlID: Word;
  itemID: Word;
  hwndItem: HWnd;
  itemData: Longint;
end;
```

The *TDeleteItemStruct* record is used to describe a deleted item from an owner-draw combo box or list box. The *wm_DeleteItem* message comes to the owner of the item, with the *lParam* parameter pointing to a *TDeleteItemStruct* record.

The *CtlType* field indicates the type of control, either list box (*odt_ListBox*) or combo box (*odt_ComboBox*). *CtlId* is the control ID of the box. *itemID* is the index of the item being deleted. *hwndItem* is the control's window handle, and *itemData* is the 32-bit value of the indexed item.

TDevMode type — WinTypes

Declaration

```
TDevMode = record
  dmDeviceName: array[0..cchDeviceName-1] of Char;
  dmSpecVersion: Word;
  dmDriverVersion: Word;
  dmSize: Word;
  dmDriverExtra: Word;
  dmFields: LongInt;
  dmOrientation: Integer;
  dmPaperSize: Integer;
  dmPaperLength: Integer;
  dmPaperWidth: Integer;
```

```
    dmScale: Integer;
    dmCopies: Integer;
    dmDefaultSource: Integer;
    dmPrintQuality: Integer;
    dmColor: Integer;
    dmDuplex: Integer;
end;
```

TDevMode records are used by *DeviceCapabilities* and *ExtDeviceMode* to hold information on a printer device driver.

The *dmDeviceName* field contains a null-terminated string giving the name of the device supported. *dmSpecVersion* is the version number of the data specification, currently $0300. *dmDriverVersion* is the version number of the driver, specified by the developer. *dmSize* is the size of the record, excluding the *dmDriverData* field at the end. *dmDriverExtra* gives the size of the *dmDriverData* field.

The *dmFields* is a 32-bit, bitmapped field, indicating which (if any) of the remaining fields have been initialized. Each bit corresponds to one field, with the constants listed in Table 4.2 defined for easy checking.

Table 4.2 *TDevMode* field flags

Flag	Field
dm_Color	*dmColor*
dm_Copies	*dmCopies*
dm_DefaultSource	*dmDefaultSource*
dm_Duplex	*dmDuplex*
dm_Orientation	*dmOrientation*
dm_PaperLength	*dmPaperLength*
dm_PaperSize	*dmPaperSize*
dm_PaperWidth	*dmPaperWidth*
dm_PrintQuality	*dmPrintQuality*
dm_Scale	*dmScale*
dm_SpecVersion	*dmSpecVersion*

The *dmOrientation* field selects the paper orientation, portrait or landscape, using one of the *dmorient_* constants.

The *dmPaperSize* field selects the paper size, using one of the *dmpaper_* constants. The *dmPaperLength* and *dmPaperWidth* fields allow the user to override the paper length and width specified in the *dmPaper* field.

The *dmScale* field scales the printed output by a factor of *dmScale*/100. A value of 75, for example, reduces images to 75% of their normal size.

The *dm_Copies* field selects the number of copies to print.

The *dmDefaultSource* field indicates the bin from which paper will feed by default, specified by one of the *dmbin_* constants. *dmPrintQuality* specifies the print resolution to be used, using one of the device-independent *dmres_* constants (which are all negative values), or a positive number, which is a device-dependent, dots-per-inch value.

The *dmColor* field selects color or monochrome printing, using the *dmcolor_* constants. *dmDuplex* selects one-sided or two-sided printing using the *dmdup_* constants.

The *dmDriverData* field contains data specific to, and defined by, the driver.

TDrawItemStruct type — WinTypes

Declaration

```
TDrawItemStruct = record
  CtlType: Word;
  CtlID: Word;
  itemID: Word;
  itemAction: Word;
  itemState: Word;
  hwndItem: HWnd;
  hDC: HDC;
  rcItem: TRect;
  itemData: Longint;
end;
```

The *TDrawItemStruct* record holds data for painting owner-draw controls. The owner of the control receives a pointer to a *TDrawItemStruct* in the *lParam* parameter of the *wm_DrawItem* message.

The *CtlType* field is the control type, designated by one of the *odt_* constants. *CtlID* is the control ID number of the control (not used for menus). *itemID* is the menu-item ID of the item index, depending on the control. This can be –1 for empty list boxes or combo boxes.

The *itemAction* field defines drawing actions, using the *oda_* constants, to specify when and how to draw the control.

The *itemState* field describes the state of the item after it is drawn, using the *ods_* constants.

The *hwndItem* field is the window handle of the control, or for a menu, the handle of the menu containing the item. *hDC* is the handle of the device context that must be used when drawing this control.

The *rcItem* field is the bounding rectangle (*TRect* record) of the control in the device context. Windows clips owner-draw controls at this boundary, but owner-draw menu items can exceed these boundaries.

The *itemData* field contains either a owner-draw list-box or combo-box value provided by the *cb_AddString*, *cb_InsertString*, *lb_AddString*, or *lb_InsertString* message that created the item, or the long integer value given to the menu item in the *NewItem* parameter of the *InsertMenu* call that inserted it. *itemData* is undefined for owner-draw buttons.

TFarProc type — WinTypes

Declaration

```
TFarProc = Pointer;
```

TFarProc is a pointer, usually to a procedure.

TGlobalHandle type — WinTypes

Declaration

```
TGlobalHandle = THandle;
```

TGlobalHandle is exactly the same as *THandle*, but you might want to use it to make clear to people reading your code that the handle is to a global item, such as a global memory block.

THandle type — WinTypes

Declaration

```
THandle = Word;
```

THandle defines a generic handle type.

THandleTable type — WinTypes

Declaration

```
THandleTable = record
  objectHandle: array[0..0] of THandle;
end;
```

THandleTable is an array of handles, usually used to store a number of drawing tools.

TLocalHandle type — WinTypes

Declaration

```
TLocalHandle = THandle;
```

TLocalHandle is exactly the same as *THandle*, but you might want to use it to make clear to people reading your code that the handle is to a local item, such as a local memory block.

TLogBrush type — WinTypes

Declaration

```
TLogBrush = record
  lbStyle: Word;
  lbColor: Longint;
  lbHatch: Integer;
end;
```

The *TLogBrush* record is used to hold the information for creation of a logical brush with the *CreateBrushIndirect* function.

The *lbStyle* field holds one of the *bs_* brush style constants, indicating that the brush should be solid, hollow, hatched, or pattern.

The *lbColor* field is a *TColorRef* record. Color is ignored if the brush style is hollow or pattern. If the style is *bs_DIBPattern*, the low-order word must contain one of the *DIB_* constants, indicating whether the colors specified are explicit, or indexed into the current palette.

The *lbHatch* field gives a hatch style. Depending on the style of the brush, *lbHatch* could contain any of the following:

Table 4.3 *TLogBrush* hatch styles

Style	lbHatch contains
bs_DIBPattern	The handle of a packed device-independent bitmap
bs_Hatched	One of the *hs_* hatch style constants, indicating the orientation of the hatch
bs_Hollow	Ignored
bs_Pattern	The handle of the bitmap defining the pattern
bs_Solid	Ignored

TLogFont type — WinTypes

Declaration

```
TLogFont = record
  lfHeight: Integer;
  lfWidth: Integer;
  lfEscapement: Integer;
  lfOrientation: Integer;
  lfWeight: Integer;
  lfItalic: Byte;
  lfUnderline: Byte;
  lfStrikeOut: Byte;
  lfCharSet: Byte;
  lfOutPrecision: Byte;
  lfClipPrecision: Byte;
  lfQuality: Byte;
  lfPitchAndFamily: Byte;
  lfFaceName: array[0..lf_FaceSize - 1] of Byte;
end;
```

The *TLogFont* record holds the attributes of a logical font for use by the *CreateFontIndirect* function.

The *lfHeight* and *lfWidth* give the average height and width of the font. *lfEscapement* and *lfOrientation* are the escapement angle and orientation angle of the text, both given in tenths of degrees, measured counterclockwise from the x-axis.

The *lfWeight* field gives the weight of the font, in inked pixels per 1000. The value may therefore be any value from 0 to 1000. 400 is considered normal, 700 bold. Actual values will vary, depending on the type face. A zero value specifies that a default weight is to be used.

The *lfItalic*, *lfUnderline*, and *lfStrikeOut* fields are normally zero. If non-zero, they signify italic, underlined, or strikeout fonts, respectively. *lfCharSet* is one of three pre-defined character sets: *ANSI_CharSet*, *OEM_CharSet*, or *Symbol_CharSet*. Other character sets may be defined.

The *lfOutPrecision* field contains one of the *out_* font precision flags; By default, it is *out_Default_Precis*. *lfClipPrecision* specifies the clipping precision of the font, defined by the *clip_* font precision flags; The default value is *clip_Default_Precis*.

The *lfQuality* field contains one of the font quality flags: *Default_Quality*, *Draft_Quality*, or *Proof_Quality*. *lfPitchAndFamily* is a combination of a font pitch flag (either *Default_Pitch*, *Fixed_Pitch*, or *Variable_Pitch*) and a font family flag (such as *ff_Roman* or *ff_Script*). *lfFaceName* holds the name of the font in a null-terminated string. A **nil** in *lfFaceName* causes GDI to use a default typeface.

TLogPalette type — WinTypes

Declaration

```
TLogPalette = record
  palVersion: Word;
  palNumEntries: Word;
  palPalEntry: array[0..0] of TPaletteEntry;
end;
```

The *TLogPalette* record holds data to define a logical palette, as used by the *CreatePalette* function.

The *palVersion* field gives the version of Windows for the structure, currently $0300. *palNumEntries* is the number of entries in the palette. *palPalEntry* is an array of *TPaletteEntry* records, one element for each of the entries in the palette.

TLogPen type — WinTypes

Declaration

```
TLogPen = record
  lopnStyle: Word;
  lopnWidth: TPoint;
  lopnColor: Longint;
end;
```

The *TLogPen* record holds the attributes of a logical pen, and is used by the *CreatePenIndirect* function.

The *lopnStyle* field holds the pen style, one of the *ps_* constants. *lopnWidth* is the width of the pen in logical units. *lopnColor* is the pen color.

TMDICreateStruct type — WinTypes

Declaration

```
TMDICreateStruct = record
  szClass: PChar;
  szTitle: PChar;
  hOwner: THandle;
  x, y: Integer;
  cx, cy: Integer;
  style: LongInt;
  lParam: LongInt;
end;
```

The *TMDICreateStruct* record hold data for the creation of an MDI child window. The *lParam* parameter of a *wm_Create* message holds a *TCreateStruct* record, the *lpCreateParams* field of which points to a *TMDICreateStruct* record, provided by a *wm_MDICreate* message.

The *szClass* field points to the child window's class. *szTitle* points to the window title. *hOwner* is the instance handle of the application that creates the window.

x nd *y* are the initial x- and y-coordinates of the child window. *cx* and *cy* are the initial width and height. A value of *cw_UseDefault* for any of these results in a default value for that coordinate, width, or height, as appropriate.

The *style* field holds additional styles for the window, which may be any of *ws_Minimize*, *ws_Maximize*, *ws_HScroll*, or *ws_VScroll*.

lParam is defined by the application.

TMeasureItemStruct type — WinTypes

Declaration

```
TMeasureItemStruct = record
  CtlType: Word;
  CtlID: Word;
  itemID: Word;
  itemWidth: Word;
  itemHeight: Word;
  itemData: Longint;
end;
```

The *TMeasureItemStruct* record holds the dimensions of an owner-draw control. A *wm_MeasureItem* message holds a pointer to a *TMeasureItemStruct* record in its *lParam* parameter. If *TMeasureItem* fields are not properly filled, owner-draw controls will not work correctly.

The *CtlType* field is the control type, designated by one of the *odt_* constants. *CtlID* is the control ID number of the control (not used for menus). *itemID* is the menu-item ID of the item index, depending on the control.

The *itemWidth* and *itemHeight* fields hold the width and height of the item, respectively.

The *itemData* field contains either a owner-draw list-box or combo-box value provided by the *cb_AddString*, *cb_InsertString*, *lb_AddString*, or *lb_InsertString* message that created the item, or the long integer value given to the menu item in the *NewItem* parameter of the *InsertMenu* call that inserted it. *itemData* is undefined for owner-draw buttons.

TMenuItemTemplateHeader type — WinTypes

Declaration

```
TMenuItemTemplateHeader = record
  versionNumber: Word;
  offset: Word;
end;
```

The *TMenuItemTemplateHeader* record

TMetaFilePict type — WinTypes

Declaration

```
TMetaFilePict = record
  mm: Integer;
  xExt: Integer;
  yExt: Integer;
  hMF: THandle;
end;
```

The *TMetaFilePict* record defines the format of the metafile picture used to exchange metafile data through the Clipboard.

The *mm* field holds the mapping mode in which the picture was drawn. *xExt* and *yExt* are the width and height of the rectangle in which the picture is drawn, unless the mapping mode is *mm_Isotropic* or

mm_Anisotropic, in which case the fields contain suggested sizes (*mm_Anitostropic*) or relative sizes (*mm_Isotropic*). *hMF* is a handle of a memory metafile.

TMetaHeader type — WinTypes

Declaration

```
TMetaHeader = record
  mtType: Word;
  mtHeaderSize: Word;
  mtVersion: Word;
  mtSize: Longint;
  mtNoObjects: Word;
  mtMaxRecord: Longint;
  mtNoParameters: Word;
end;
```

The *TMetaHeader* record defines the format of the header of a metafile. A metafile consists of a *TMetaHeader* record, followed by a list of metafile records, usually of type *TMetaRecord*.

The *mtType* field holds one of two values: 1 indicates that the metafile is in memory, and 2 indicates that the metafile is on a disk.

The *mtHeaderSize* field gives the size of the header, in bytes. *mtVersion* is the Windows version number, currently $0300 for version 3.0. *mtSize* is the size, in words, of the file.

The *mtNoObjects* and *mtMaxRecord* fields indicate the maximum number of objects the metafile can hold and the size (in words) of the largest metafile record.

mtNoParameters is not currently used.

TMetaRecord type — WinTypes

Declaration

```
TMetaRecord = record
  rdSize: Longint;
  rdFunction: Word;
  rdParm: array[0..0] of Word;
end;
```

The *TMetaRecord* record defines a typical metafile record. A list of such records follows the header of a metafile.

The *rdSize* field gives the size of the record, in words. *rdFunction* is the function number, specified by a *meta_* constant. *rdParm* is an array of *Word*-type parameters for the function, stored in the reverse order that they will be passed to the function.

TMsg type — WinTypes

Declaration

```
TMsg = record
  hwnd: HWnd;
  message: Word;
  wParam: Word;
  lParam: LongInt;
  time: Longint;
  pt: TPoint;
end;
```

The *TMsg* record holds message data that gets dispatched to applications by Windows. The information is routed to the appropriate elements (windows) for processing. The various fields contain important information for the application. All Windows messages, as well as their parameters, are listed in Chapter 3, "Windows message reference."

The *hwnd* field is the handle of the window that receives the message. *message* is the number of the message.

The *wParam* and *lParam* fields hold word-length and long-integer-length information for the window. This information varies depending on the value of *message*. Message parameter values are described for each message in Chapter 3.

The *time* field holds the time at which the message was posted, and *pt* holds the screen position of the cursor when the message was posted.

TMultiKeyHelp type — WinTypes

Declaration

```
TMultiKeyHelp = record
  mkSize: Word;
  mkKeyList: Byte;
  szKeyPhrase: array[0..0] of Byte;
end;
```

The *TMultiKeyHelp* record holds an index to a table of key words and a phrase to be searched for, for use by the Windows help system.

The *mkSize* field holds the length of the record. *mkKeyList* is a character that indicates which key-word table should be searched. *szKeyPhrase* is a mull-terminated string holding the key word to be located.

TOFStruct type — WinTypes

Declaration

```
TOFStruct = record
  cBytes: Byte;
  fFixedDisk: Byte;
  nErrCode: Word;
  reserved: array[0..3] of Byte;
  szPathName: array[0..127] of Char;
end;
```

The *TOFStruct* record holds information on a file, read when the file was opened.

The *cBytes* field holds the length of the record. *fFixedDisk* is non-zero if the file is on a fixed (hard) disk; zero otherwise. *nErrCode* holds the DOS error code if the *OpenFile* function failed (*OpenFile* returns –1 if it fails.). *reserved* holds four bytes, reserved for future Windows use.

The *szPathName* field is a null-terminated string holding the full path name of the file.

TPaintStruct type — WinTypes

Declaration

```
TPaintStruct = record
  hdc: HDC;
  fErase: Bool;
  rcPaint: TRect;
  fRestore: Bool;
  fIncUpdate: Bool;
  rgbReserved: array[0..15] of Byte;
end;
```

The *TPaintStruct* record holds information used by an application for painting the client areas of its windows. Most of the information is reserved for internal use by Windows, but several fields can be used by the user.

The *hdc* field is the handle of the display context on which the painting is to occur. *fErase* indicates whether the background has been redrawn; a

zero indicates it has not been redrawn. *rcPaint* defines the rectangle in which painting is to take place.

All the other fields are reserved for internal Windows use.

TPaletteEntry type — WinTypes

Declaration

```
TPaletteEntry = record
  peRed: Byte;
  peGreen: Byte;
  peBlue: Byte;
  peFlags: Byte;
end;
```

The *TPaletteEntry* record type defines an entry in a logical palette, such as that defined by *TLogPalette*.

The *peRed*, *peGreen*, and *peBlue* fields represent the intensities of red, green, and blue, respectively, in the palette entry. *peFlags* contains information on how the palette is to be used. It can be zero or one of the *pc_* flags: *pc_Explicit*, *pc_NoCollapse*, or *pc_Reserved*.

TPattern type — WinTypes

Declaration

```
TPattern = TLogBrush;
```

TPattern is another name for *TLogBrush*. When a logical brush is used for pattern-filling, you might want to use the *TPattern* name for clarity.

TPoint type — WinTypes

Declaration

```
TPoint = record
  x: Integer;
  y: Integer;
end;
```

The *TPoint* record is very simple, but it is also very useful. It specifies the x- and y-coordinates (in the *x* and *y* fields, respectively) of a point on the screen or in a window.

TRect type — WinTypes

Declaration

```
TRect = record
  left: Integer;
  top: Integer;
  right: Integer;
  bottom: Integer;
end;
```

The *TRect* record defines a rectangular region, given its upper left and lower right corners. The *left* and *top* fields are the x- and y-coordinates of the upper left corner of the rectangle. *right* and *bottom* are the x- and y-coordinates of the lower right corner.

Note that a rectangle may not exceed 32,768 units in width or height.

TRGBQuad type — WinTypes

Declaration

```
TRGBQuad = record
  rgbBlue: Byte;
  rgbGreen: Byte;
  rgbRed: Byte;
  rgbReserved: Byte;
end;
```

The *TRGBQuad* record holds RGB color data for use by bitmaps, such as in the *bmiColors* field of a *TBitmapInfo* record.

The *rgbBlue*, *rgbGreen*, and *rgbRed* fields represent the intensities of blue, green, and red, respectively, in the bitmap pixel entry. *rgbReserved* is not used, and must be zero.

TRGBTriple type — WinTypes

Declaration

```
TRGBTriple = record
  rgbtBlue: Byte;
  rgbtGreen: Byte;
  rgbtRed: Byte;
end;
```

The *TRGBTriple* record specifies RGB color intensities for bitmaps, such as in the *bmciColors* field of a *TBitmapCoreInfo* record.

The *rgbtBlue*, *rgbtGreen*, and *rgbtRed* fields represent the intensities of blue, green, and red, respectively, in the bitmap pixel entry.

TTextMetric type — WinTypes

Declaration

```
TTextMetric = record
  tmHeight: Integer;
  tmAscent: Integer;
  tmDescent: Integer;
  tmInternalLeading: Integer;
  tmExternalLeading: Integer;
  tmAveCharWidth: Integer;
  tmMaxCharWidth: Integer;
  tmWeight: Integer;
  tmItalic: Byte;
  tmUnderlined: Byte;
  tmStruckOut: Byte;
  tmFirstChar: Byte;
  tmLastChar: Byte;
  tmDefaultChar: Byte;
  tmBreakChar: Byte;
  tmPitchAndFamily: Byte;
  tmCharSet: Byte;
  tmOverhang: Integer;
  tmDigitizedAspectX: Integer;
  tmDigitizedAspectY: Integer;
end;
```

The *TTextMetric* record holds numerous fields describing a physical font in units that depend on the display context's mapping mode. *TTextMetric* records are used by the *GetDeviceCaps* and *GetTextMetrics* functions.

The *tmHeight* field is the height of characters in the font, equal to the sum of the ascent (*tmAscent*) above the baseline and descent (*tmDescent*) below the baseline.

The *tmInternalLeading* and *tmExternalLeading* fields specify the amount of space allowed beyond the range of *tmHeight*. *tmInternalLeading* is space inside the boundary; *tmExternalLeading* is added between rows of text. Either field may be set to zero.

The *tmAveCharWidth* and *tmMaxCharWidth* give the average and maximum character widths of characters in the font.

tmWeight is the weight of the font.

tmItalic, *tmUnderlined*, and *tmStruckOut*, if non-zero, indicate italic, underlined, and strikeout text, respectively.

The range of defined characters in the font is given by *tmFirstChar* and *tmLastChar*, with characters outside that range having the character specified in *tmDefaultChar* substituted for them. *tmBreakChar* is the character that delineates word breaks for justification purposes.

The pitch, family, and character set of the font are defined by *tmPitchAndFamily* and *tmCharSet*. The lowest-order bit in *tmPitchAndFamily* determines the pitch of the font: fixed if the bit is clear, variable if the bit is set. The four high-order bits determine the font family, which can be set or checked using the *ff_* font family flags. The character set is set in *tmCharSet*, using the font character set flags.

The *tmOverhang* field holds the extra width added per string to some fonts that have to be synthesized, such as when making a bold font from a normal font.

The horizontal and vertical aspects of the device for which the font was designed are in the *tmDigitizedAspectX* and *tmDigitizedAspectY* fields, respectively.

TWndClass type — WinTypes

Declaration

```
TWndClass = record
  style: Word;
  lpfnWndProc: TFarProc;
  cbClsExtra: Integer;
  cbWndExtra: Integer;
  hInstance: THandle;
  hIcon: HIcon;
```

```
  hCursor: HCursor;
  hbrBackground: HBrush;
  lpszMenuName: PChar;
  lpszClassName: PChar;
end;
```

The *TWndClass* record holds the attributes of a window class, also known as the registration attributes, as they are registered with the *RegisterClass* function.

The *style* field holds the class style. It can hold one or a combination of the *cs_* class style constants.

The *lpfnWndProc* field points to the window's window function, the routine that receives and processes messages.

cbClsExtra is the number of bytes to be allocated at the end of the *TWndClass* record. These are called the class' extra bytes, and can be accessed with the *GetWindowLong* or *GetWindowWord* functions, or set with the *SetWindowLong* or *SetWindowWord* functions.

cbWndExtra gives the number of bytes to allocate at the end of the window instance.

hInstance is an instance handle that must indicate the class module. It must not be zero.

The *hIcon*, *hCursor*, and *hbrBackground* fields are the handles of the class' icon and cursor, and the class' background color, respectively. The background color should be a color value (one of the standard system colors, given by a *color_* constant, incremented by one) or the handle of a brush for painting the background. If *hbrBackground* is zero, the application's background must be painted whenever its client area is painted. The need for this can be determined by processing the *wm_EraseBkgnd* message or by checking the *fErase* field of the *TPaintStruct* record created by *BeginPaint*.

The *lpszMenuName* and *lpszClassName* field both point to null-terminated strings, being the resource name of the class menu and the name of the class, respectively.

PART 2

ObjectWindows Reference

C H A P T E R

5

Object Windows reference

This chapter contains an alphabetical listing of all the standard ObjectWindows object types, with explanations of their general purposes and usage, their fields, methods and color palettes.

To find information on a specific object, keep in mind that many of the properties of the objects in the hierarchy are inherited from ancestor objects. Rather than duplicate all that information endlessly, this chapter only documents fields and methods that are *new* or *changed* for a particular object.

To save you some hunting, all fields and methods are indexed.

For example, if you want to know about the *Parent* field of a *TEdit* object, you might first look under *TEdit*'s fields, where you won't find *Parent* listed. You would then check *TEdit*'s immediate ancestor in the hierarchy, *TStatic*. Again, *Parent* will not be listed. You would next check *TStatic*'s immediate ancestor, *TControl*, and so on, until you got to *TWindowsObject*. There you will find complete information about *Parent*, which is inherited unchanged by *TEdit*.

Each object's entry is laid out in the following format:

TSample object — Object's unit

First is a general overview of the object, it's relationship with other objects, and general usage.

Fields

This section lists all fields for each object, alphabetically. In addition to showing the declaration of the field and an explanation of its use, there is a **Read only** or **Read/write** designation. Read-only fields are generally fields that are set up and maintained by the object's methods, and they should *not* be on the left side of an assignment statement.

AField `AField: SomeType;` **Read only**

AField is a field that holds some information about this sample object. This text explains how it functions, what it means, and how you use it.

See also: related fields, methods, objects, global functions, etc.

AnotherField `AnotherField: Word;` **Read/write**

AnotherField has similar information to that for *AField*.

Methods

This section lists all methods which are either newly defined for this object or which override inherited methods. For virtual methods, an indication will be given as to how often you will probably need to override the method: Never, Seldom, Sometimes, Often, or Always.

Init `constructor Init(AParameter: SomeType);`

Init creates a new sample object, setting the *AField* field to *AParameter*.

Zilch `procedure Zilch; virtual;`

Override: Sometimes

The *Zilch* procedure causes the sample object to perform some action.

See also: *TSomethingElse.Zilch*

TApplication WObjects

TApplication provides the structure for a ObjectWindows applications. All ObjectWindows applications will derive an object type from *TApplication* primarily to construct a main window of a user-defined object type.

Fields

HAccTable `HAccTable: THandle;` **Read/write**

HAccTable holds a handle to a Windows accelerator table resource defined for the application.

KBHandlerWnd `KBHandlerWnd: PWindowsObject;` **Read only**

KBHandlerWnd points to the currently active window if that window's keyboard handler mechanism is enabled. This mechanism allows a window with controls to process keyboard input like a dialog. *KBHandlerWnd* is **nil** if the mechanism is disabled for the active window.

MainWindow `MainWindow: PWindowsObject;` **Read/write**

MainWindow points to the application's main, overlapped window, which should be instantiated by your application type's *InitMainWindow* method.

Name `Name: PChar;`

Name holds the name of the application. This can be used internally by the application's code.

Status `Status: Integer;`

Status indicates the current state of running application. It is running successfully if *Status* is greater than or equal to zero. Error values include *em_InvalidMainWindow* for an invalid main window object.

Methods

Init `constructor Init(AName: PChar);`

Override: Sometimes

Constructs the application object. First calls *TObject.Init*, then sets the global variable *Application* to *@Self*, sets the *Name* field to *AName*, sets the *HAccTable* and *Status* fields to zero, and initializes the *MainWindow* and *KBHandlerWnd* fields to **nil**.

If this is the first running instance of this application, *Init* calls *InitApplication*. If *InitApplication* succeeds (that is, the *Status* field is still zero), *InitInstance* is called.

You may override this method to, for example, load an accelerator table for your application. Make sure you call this method from any method which overrides it.

See also: *TApplication.InitApplication, TApplication.InitInstance, TObject.Init*

Done

```
destructor Done; virtual;
```

Override: Sometimes

Disposes of the application's owned objects by disposing *MainWindow*, then calls *TObject.Done* to terminate the application.

See also: *TObject.Done*

CanClose

```
function CanClose: Boolean; virtual;
```

Override: Seldom

Returns *True* if it is OK for the application to close. As a default, it calls the *CanClose* method of its main window and returns its return value. This method will seldom be overridden; closing behavior can be overridden in the main window's *CanClose* method.

See also: *TWindowsObject.CanClose, TWindowsObject.WMDestroy*

Error

```
procedure Error(ErrorCode: Integer); virtual;
```

Override: Sometimes

Error processes errors identified by the error value passed in *ErrorCode*. These errors can be generated by the application object or any window or dialog object, and *ErrorCode* can be one of the following errors detected and reported by ObjectWindows, or an error you define:

em_InvalidWindow
em_OutOfMemory
em_InvalidClient
em_InvalidChild
em_InvalidMainWindow

The *em_* constants are described in Chapter 6, "Global reference."

Error displays the error code in a message box and asks the user if it is okay to proceed. If not, program execution stops.

ExecDialog

```
function ExecDialog(ADialog: PWindowsObject): Integer; virtual;
```

Override: Never

Executes the modal dialog object passed in *ADialog*, after checking *ValidWindow*, by calling the dialog object's *Execute* method. If memory is low, or if the dialog cannot be executed, *ExecDialog* disposes the object and returns a negative error status.

See also: *TDialog.Execute*

InitApplication

```
procedure InitApplication; virtual;
```

Override: Sometimes

Performs any initialization necessary only for the first executing instance of the application. *TApplication.InitApplication* does nothing. Your application object type may override *InitApplication* to perform application-specific initialization.

InitInstance

```
procedure InitInstance; virtual;
```

Override: Sometimes

Performs any initialization necessary for every executing instance of the application. *TApplication.InitInstance* calls *InitMainWindow*, and creates and shows the main window element by calling *MakeWindow* and *Show*. If the main window could not be created, the *Status* field is set to *em_InvalidMainWindow*. If you override this method, be sure to explicitly call *TApplication.InitInstance*.

See also: *TApplication.InitMainWindow*

InitMainWindow

```
procedure InitMainWindow; virtual;
```

Override: Always

By default, *TApplication.InitMainWindow* instantiates a generic *TWindow* object with no title. Override *InitMainWindow* to construct a useful main window object and store it in *MainWindow*. Typical use:

```
procedure MyApplication.InitMainWindow;
begin
  MainWindow := New(PMyWindow, Init('Window Caption'));
end;
```

MakeWindow

```
function MakeWindow(AWindowsObject: PWindowsObject): PWindowsObject;
virtual;
```

Override: Never

Attempts to create a window or modeless dialog element associated with the object passed in *AWindowsObject*, after performing checks of safety pool usage. If memory is low (*LowMemory* returns *True*), or if the window or dialog cannot be created, *MakeWindow* disposes the object and returns **nil**. If successful, it returns *AWindowsObject*.

See also: *TWindow.Create, LowMemory*

MessageLoop

```
procedure MessageLoop; virtual;
```

Override: Never

Operates the application's general message loop which runs during the lifetime of the application. *TApplication.MessageLoop* calls *ProcessAppMsg* to handle special messages for modeless dialogs, accelerators and MDI accelerators. Any non-standard message processing should be done in *ProcessAppMsg*, rather than in *MessageLoop*.

See also: *TApplication.ProcessAppMsg*

ProcessAppMsg `function ProcessAppMsg(var Message: TMsg): Boolean; virtual;`

Override: Sometimes

Checks for special processing for modeless dialog, accelerator and MDI accelerator messages. Calls *ProcessDlgMsg*, *ProcessMDIAccels*, and *ProcessAccels* and returns *True* if any of these special messages are encountered. If your application does not create modeless dialogs, does not respond to accelerators, and is not an MDI application, you can improve performance by overriding this method to simply return *False*.

ProcessDlgMsg `function ProcessDlgMsg(var Message: TMsg): Boolean; virtual;`

Override: Sometimes

Handles special modeless dialog and window message processing for handling keyboard input for controls. If your application creates no modeless dialogs or windows with controls, you can improve performance by overriding this method to simply return *False*.

ProcessAccels `function ProcessAccels(var Message: TMsg): Boolean; virtual;`

Override: Sometimes

Handles special accelerator message processing. If your application's windows do not respond to accelerators, you can improve performance by overriding this method to simply return *False*.

ProcessMDIAccels `function ProcessMDIAccels(var Message: TMsg): Boolean; virtual;`

Handles special accelerator message processing for MDI-compliant applications. If your application is not an MDI application, you can improve performance by overriding this method to simply return *False*.

Run `procedure Run; virtual;`

Override: Seldom

Sets the application in motion by calling *MessageLoop* if initialization was successful (that is, the *Status* field is zero).

See also: *TApplication.MessageLoop*

SetKBHandler `procedure SetKBHandler(AWindowsObject: PWindowsObject);`

Override: Never

Activates keyboard handling (translation of keyboard input into control selections) for the given window by setting *KBHandlerWnd* to *AWindowsObject*.

See also: *TApplication.KBHandlerWnd*

ValidWindow `function ValidWindow(AWindowsObject: PWindowsObject): PWindowsObject;`

Determines whether *AWindowsObject* is a valid object. If *AWindowsObject* is valid, *ValidWindow* returns a pointer to it, otherwise it returns **nil**. *AWindowsObject* is invalid if either of two conditions occurs: allocation of

the object ate into the safety pool (*LowMemory* is *True*), or the *Status* field of *AWindowsObject* is non-zero.

See also: *LowMemory, TWindowsObject.Status*

TBufStream WObjects

TBufStream implements a buffered version of *TDosStream*. The additional fields specify the size and location of the buffer, together with the current and last positions within the buffer. In addition to overriding the eight methods of *TDosStream*, *TBufStream* defines the abstract *TStream.Flush* method. The *TBufStream* constructor creates and opens a named file by calling *TDosStream.Init*, then creates the buffer with *GetMem*.

TBufStream is significantly more efficient than *TDosStream* when a large number of small data transfers take place on the stream, such as when loading and storing objects using *TStream.Get* and *TStream.Put*.

Fields

Buffer `Buffer: Pointer;` **Read only**

A pointer to the start of the stream's buffer

BufSize `BufSize: Word;` **Read only**

The size of the buffer in bytes

BufPtr `BufPtr: Word;` **Read only**

An offset from the *Buffer* pointer indicating the current position within the buffer.

BufEnd `BufEnd: Word;` **Read only**

If the buffer is not full, *BufEnd* gives an offset from the *Buffer* pointer to the last used byte in the buffer.

Methods

Init `constructor Init(FileName: FNameStr; Mode, Size: Word);`

Creates and opens the named file with access mode *Mode* by calling *TDosStream.Init*. Also creates a buffer of *Size* bytes with a *GetMem* call. The

Handle, *Buffer* and *BufSize* fields are suitably initialized. Typical buffer sizes range from 512 bytes to 2,048 bytes.

See also: *TDosStream.Init*

Done `destructor Done; virtual;`

Override: Never Closes and disposes of the file stream; flushes and disposes of its buffer.

See also: *TBufStream.Flush*

Flush `procedure Flush; virtual;`

Override: Never Flushes the calling file stream's buffer provided the stream is *stOK*.

See also: *TBufStream.Done*

GetPos `function GetPos: Longint; virtual;`

Override: Never Returns the value of the calling stream's current position (not to be confused with *BufPtr*, the current location within the buffer).

See also: *TBufStream.Seek*

GetSize `function GetSize: Longint; virtual;`

Override: Never Flushes the buffer then returns the total size in bytes of the calling stream.

Read `procedure Read(var Buf; Count: Word); virtual;`

Override: Never If *stOK*, reads *Count* bytes into the *Buf* buffer starting at the calling stream's current position.

Note that *Buf* is *not* the stream's buffer, but an external buffer to hold the data read in from the stream.

See also: *TBufStream.Write*, *stReadError*

Seek `procedure Seek(Pos: Longint); virtual;`

Override: Never Flushes the buffer then resets the current position to *Pos* bytes from the start of the calling stream. The start of a stream is position 0.

See also: *TBufStream.GetPos*, *TBufStream.GetPos*

Truncate `procedure Truncate; virtual;`

Override: Never Flushes the buffer then deletes all data on the calling stream from the current position to the end. The current position is set to the new end of the stream.

See also: *TBufStream.GetPos*, *TBufStream.Seek*

Write `procedure Write(var Buf; Count: Word); virtual;`

Override: Never

If *stOK*, writes *Count* bytes from the *Buf* buffer to the calling stream, starting at the current position.

Note that *Buf* is *not* the stream's buffer, but an external buffer to hold the data being written to the stream. When *Write* is called, *Buf* will point to the variable whose value is being written.

See also: *TBufStream.Read, stWriteError*

TButton — WObjects

TButton is an interface object that represents a corresponding push button element in Windows. *TButton* objects are not normally used in dialog boxes (*TDialog*) or dialog windows (*TDlgWindow*), but are used when you want to display a standalone button as a child window in another window's client area.

There are two types of push buttons. A regular button appears with a thin border. A default button appears with a thick border and represents the default action of the window. There can only be one default push button in a window.

Methods

Init

```
constructor Init(AParent: PWindowsObject; AnId: Integer; AText: PChar;
  X, Y, W, H: Integer; IsDefault: Boolean);
```

Constructs a button object with the passed parent window (*AParent*), control ID (*AnId*); associated text (*AText*); position (*X*, *Y*), relative to the origin of the parent window's client area; width (*W*); and height (*H*). Calls *TControl.Init* and then adds *bs_DefPushButton* to the *Attr.Style* field if *IsDefault* is *True*, else adds *bs_PushButton*.

See also: *TControl.Init*

InitResource

```
constructor InitResource(AParent: PWindowsObject, ResourceID: Word);
```

Associates the button by constructing an ObjectWindows object to correspond to a button element created by a dialog resource definition. Calls *TControl.InitResource* and *DisableTransfer* to exclude the button from the transfer mechanism, since they have no data to be transferred.

See also: *TControl.InitResource, TWindowsObject.DisableTransfer*

GetClassName `function GetClassName: PChar; virtual;`

Override: Never Returns the name of *TButton's* window class, 'Button'.

TCheckBox — WObjects

TCheckBox is an interface object that represents a corresponding check box element in Windows. *TCheckBox* objects are not normally used in dialog boxes (*TDialog*) or dialog windows (*TDlgWindow*), but are used when you want to display a standalone check box as a child window in another window's client area.

Check boxes have two states: checked and unchecked. *TCheckBox* methods are concerned primarily with managing the check box's state. Optionally, a check box can be part of a group (*TGroupBox*) which visually and functionally groups its controls.

Field

Group `Group: PGroupBox;` **Read only**

Group points the *TGroupBox* control object that unifies the check box with other check boxes and radio buttons (*TRadioButton*). If the check box is not part of a group, *Group* is equal to **nil**.

See also: *TGroupBox, TRadioButton*

Methods

Init

```
constructor Init(AParent: PWindowsObject; AnID: Integer; ATitle:
  PChar; X, Y, W, H: Integer; AGroup: PGroupBox);
```

Override: Sometimes Constructs a check box object with the passed parent window (*AParent*), control ID (*AnId*), associated text (*ATitle*), position (*X, Y*) relative to the origin of the parent window's client area, width (*W*), height (*H*), and associated group box (*AGroup*). *TCheckBox.Init* sets the check box's *Attr.Style* field to *ws_Child* **or** *ws_Visible* **or** *ws_TabStop* **or** *bs_AutoCheckBox*.

InitResource `constructor InitResource(AParent: PWindowsObject; ResourceID: Word);`

Associates a *TCheckBox* object with the resource given by *ResourceID* by calling *TButton.InitResource*, then enabling the transfer mechanism by calling *EnableTransfer*.

Load `constructor Load(var S: TStream);`

Constructs and loads a check box from the stream *S* by first calling *TButton.Load* and then reading the additional field (*Group*) introduced by *TCheckBox*.

See also: *TControl.Load*

BNClicked

```
procedure BNClicked(var Msg: TMessage); virtual nf_First + bn_Clicked;
```

Override: Sometimes

Automatically responds to notification messages indicating that the check box was clicked by toggling its state. If the check box's *Group* is not **nil**, *TCheckBox.BNClicked* notifies the *TGroupBox* by calling its *SelectionChanged* method.

See also: *TGroupBox.SelectionChanged*

Check

```
procedure Check; virtual;
```

Override: Seldom

Forces the check box into the checked state by calling *SetCheck*.

See also: *TCheckBox.SetCheck*

GetCheck

```
function GetCheck: Word; virtual;
```

Override: Seldom

Returns *bf_Unchecked* (zero) if the check box is unchecked, *bf_Checked* (1) if it is checked, or *bf_Grayed* (2) if it is grayed.

SetCheck

```
procedure SetCheck(CheckFlag: Word); virtual;
```

Override: Seldom

Forces the check box into the state specified by *CheckFlag*. If *CheckFlag* is zero, the state will be unchecked. If it is one, the state will be checked. If it is 2, the state will be grayed. *SetCheck* also informs the check box's group that the selection has changed.

See also: *TGroupBox.SelectionChanged*

Store

```
procedure Store(var S: TStream);
```

Stores the check box on the stream *S* by first calling *TControl.Store* and then writing the additional field (*Group*) introduced by *TCheckBox*.

See also: *TControl.Store*

Toggle

```
procedure Toggle; virtual;
```

Override: Seldom

Toggles the state of the check box by calling *Check* or *Uncheck*. For a 3-state check box, toggles through all three states: checked, unchecked, and grayed.

See also: *TCheckBox.Check*, *TCheckBox.Uncheck*

Transfer

```
function Transfer(DataPtr: Pointer; TransferFlag: Word): Word;
  virtual;
```

Override: Sometimes

Transfers the state of the check box in a *Word*-type value (*bf_Checked* if checked, *bf_Unchecked* if unchecked) to or from the memory location pointed to by *DataPtr*. If *TransferFlag* is *tf_GetData*, the check box's state data is transferred to the memory location. If *TransferFlag* is *tf_SetData*, the check box is set to the state indicated in the memory location. *Transfer* returns the number of bytes stored in or retrieved from the memory location. If you pass *tf_SizeData*, *Transfer* returns the size of the transfer data, which is two bytes.

Uncheck `procedure Uncheck; virtual;`

Override: Seldom

Forces the check box into the unchecked state by calling *SetCheck*.

See also: *TCheckBox.SetCheck*

TCollection — WObjects

TCollection is an abstract type for implementing any collection of items, including other objects. *TCollection* is a more general concept than the traditional array, set, or list. *TCollection* objects size themselves dynamically at run time and offer a base type for many specialized types such as *TSortedCollection* and *TStrCollection*. In addition to methods for adding and deleting items, *TCollection* offers several *iterator* routines that call a procedure or function for each item in the collection.

Fields

Items `Items: PItemList;` **Read only**

A pointer to an array of item pointers.

See also: *TItemList* type

Count `Count: Integer;` **Read only**

The current number of items in the collection, up to *MaxCollectionSize*.

See also: *MaxCollectionSize* variable

Limit `Limit: Integer;` **Read only**

The currently allocated size (in elements) of the *Items* list.

See also: *Delta*, *TCollection.Init*

Delta `Delta: Integer;` **Read only**

The number of items by which to increase the *Items* list whenever it becomes full. If *Delta* is zero, the collection cannot grow beyond the size set by *Limit*.

Increasing the size of a collection is fairly costly in terms of performance. To minimize the number of times it has to occur, try to set the initial *Limit* to an amount that will encompass all the items you might want to collect, and set *Delta* to a figure that will allow a reasonable amount of expansion.

See also: *Limit, TCollection.Init*

Methods

Init `constructor Init(ALimit, ADelta: Integer);`

Creates a collection with *Limit* set to *ALimit* and *Delta* set to *ADelta*. The initial number of items will be limited to *ALimit*, but the collection is allowed to grow in increments of *ADelta* until memory runs out or the number of items reaches *MaxCollectionSize*.

See also: *TCollection.Limit, TCollection.Delta*

Load `constructor Load(var S: TStream);`

Creates and loads a collection from the given stream. *TCollection.Load* calls *GetItem* for each item in the collection.

See also: *TCollection.GetItem*

Done `destructor Done; virtual;`

Override: Often

Deletes and disposes of all items in the collection by calling *TCollection.FreeAll* and setting *Limit* to 0

See also: *TCollection.FreeAll, TCollection.Init*

At `function At(Index: Integer): Pointer;`

Returns a pointer to the item indexed by *Index* in the collection. This method lets you treat a collection as an indexed array. If *Index* is less than zero or greater than or equal to *Count*, the *Error* method is called with an argument of *coIndexError*, and a value of **nil** is returned.

See also: *TCollection.IndexOf*

AtDelete `procedure AtDelete(Index: Integer);`

Deletes the item at the *Index*'th position and moves the following items up by one position. *Count* is decremented by 1, but the memory allocated to the collection (as given by *Limit*) is not reduced. If *Index* is less than zero or greater than or equal to *Count*, the *Error* method is called with an argument of *coIndexError*.

See also: *TCollection.FreeItem, TCollection.Free, TCollection.Delete*

AtFree

```
procedure AtFree(Index: Integer);
```

Disposes and deletes the item at the *Index*'th position.

AtInsert

```
procedure AtInsert(Index: Integer; Item: Pointer);
```

Inserts *Item* at the *Index*'th position and moves the following items down by one position. If *Index* is less than zero or greater than *Count*, the *Error* method is called with an argument of *coIndexError* and the new *Item* is not inserted. If *Count* is equal to *Limit* before the call to *AtInsert*, the allocated size of the collection is expanded by *Delta* items using a call to *SetLimit*. If the *SetLimit* call fails to expand the collection, the *Error* method is called with an argument of *coOverflow* and the new *Item* is not inserted.

See also: *TCollection.At, TCollection.AtPut*

AtPut

```
procedure AtPut(Index: Integer; Item: Pointer);
```

Replaces the item at index position *Index* with the item given by *Item*. If *Index* is less than zero or greater than or equal to *Count*, the *Error* method is called with an argument of *coIndexError*.

See also: *TCollection.At, TCollection.AtInsert*

Delete

```
procedure Delete(Item: Pointer);
```

Deletes the item given by *Item* from the collection. Equivalent to *AtDelete(IndexOf(Item))*.

See also: *TCollection.AtDelete, TCollection.DeleteAll*

DeleteAll

```
procedure DeleteAll;
```

Deletes all items from the collection by setting *Count* to zero.

See also: *TCollection.Delete, TCollection.AtDelete*

Error

```
procedure Error(Code, Info: Integer); virtual;
```

Override: Sometimes

Called whenever a collection error is encountered. By default, this method produces a run-time error of (212 – *Code*).

See also: co*XXXX* collection constants

FirstThat

```
function FirstThat(Test: Pointer): Pointer;
```

FirstThat applies a Boolean function, given by the function pointer *Test*, to each item in the collection until *Test* returns *True*. The result is the item pointer for which *Test* returned *True*, or **nil** if the *Test* function returned *False* for all items. *Test* must point to a **far** local function taking one *Pointer* parameter and returning a *Boolean* value. For example

```
function Matches(Item: Pointer): Boolean; far;
```

The *Test* function *cannot* be a global function.

Assuming that *List* is a *TCollection*, the statement

```
P := List.FirstThat(@Matches);
```

corresponds to

```
I := 0;
while (I < List.Count) and not Matches(List.At(I)) do Inc(I);
if I < List.Count then P := List.At(I) else P := nil;
```

See also: *TCollection.LastThat, TCollection.ForEach*

ForEach `procedure ForEach(Action: Pointer);`

ForEach applies an action, given by the procedure pointer *Action*, to each item in the collection. *Action* must point to a **far** local procedure taking one *Pointer* parameter. For example

```
function PrintItem(Item: Pointer); far;
```

The *Action* procedure *cannot* be a global procedure.

Assuming that *List* is a *TCollection*, the statement

```
List.ForEach(@PrintItem);
```

corresponds to

```
for I := 0 to List.Count - 1 do PrintItem(List.At(I));
```

See also: *TCollection.FirstThat, TCollection.LastThat*

Free `procedure Free(Item: Pointer);`

Deletes and disposes of the given *Item*. Equivalent to

```
Delete(Item);
FreeItem(Item);
```

See also: *TCollection.FreeItem, TCollection.Delete*

FreeAll `procedure FreeAll;`

Deletes and disposes of all items in the collection.

See also: *TCollection.DeleteAll*

FreeItem `procedure FreeItem(Item: Pointer); virtual;`

Override: Sometimes

The *FreeItem* method must dispose the given *Item*. The default *TCollection.FreeItem* assumes that *Item* is a pointer to a descendant of *TObject*, and thus calls the *Done* destructor:

```
if Item <> nil then Dispose(PObject(Item), Done);
```

FreeItem is called by *Free* and *FreeAll*, but it should never be called directly.

See also: *TCollection.Free, TCollection.FreeAll*

GetItem `function TCollection.GetItem(var S: TStream): Pointer; virtual;`

Override: Sometimes

Called by *TCollection.Load* for each item in the collection. This method can be overridden but should not be called directly. The default *TCollection.GetItem* assumes that the items in the collection are descendants of *TObject*, and thus calls *TStream.Get* to load the item:

```
GetItem := S.Get;
```

See also: *TStream.Get, TCollection.Load, TCollection.Store*

IndexOf `function IndexOf(Item: Pointer): Integer; virtual;`

Override: Never

Returns the index of the given *Item*. The converse operation to *TCollection.At*. If *Item* is not in the collection, *IndexOf* returns –1.

See also: *TCollection.At*

Insert `procedure Insert(Item: Pointer); virtual;`

Override: Never

Inserts *Item* into the collection, and adjusts other indexes if necessary. By default, insertions are made at the end of the collection by calling *AtInsert(Count, Item);*

See also: *TCollection.AtInsert*

LastThat `function LastThat(Test: Pointer): Pointer;`

LastThat applies a Boolean function, given by the function pointer *Test*, to each item in the collection in reverse order until *Test* returns *True*. The result is the item pointer for which *Test* returned *True*, or **nil** if the *Test* function returned *False* for all items. *Test* must point to a **far** local function taking one *Pointer* parameter and returning a *Boolean*, for example

```
function Matches(Item: Pointer): Boolean; far;
```

The *Test* function *cannot* be a global function.

Assuming that *List* is a *TCollection*, the statement

```
P := List.LastThat(@Matches);
```

corresponds to

```
I := List.Count - 1;
while (I >= 0) and not Matches(List.At(I)) do Dec(I);
if I >= 0 then P := List.At(I) else P := nil;
```

See also: *TCollection.FirstThat, TCollection.ForEach*

Pack

```
procedure Pack;
```

Deletes all **nil** pointers in the collection.

See also: *TCollection.Delete, TCollection.DeleteAll*

PutItem

```
procedure PutItem(var S: TStream; Item: Pointer); virtual;
```

Override: Sometimes

Called by *TCollection.Store* for each item in the collection. This method can be overridden but should not be called directly. The default *TCollection.PutItem* assumes that the items in the collection are descendants of *TObject*, and thus calls *TStream.Put* to store the item:

```
S.Put(Item);
```

See also: *TCollection.GetItem, TCollection.Store, TCollection.Load*

SetLimit

```
procedure SetLimit(ALimit: Integer); virtual;
```

Override: Seldom

Expands or shrinks the collection by changing the allocated size to *ALimit*. If *ALimit* is less than *Count* it is set to *Count*, and if *ALimit* is greater than *MaxCollectionSize* it is set to *MaxCollectionSize*. Then, if *ALimit* is different from the current *Limit*, a new *Items* array of *ALimit* elements is allocated, the old *Items* array is copied into the new array, and the old array is disposed.

See also: *TCollection.Limit, TCollection.Count, MaxCollectionSize* variable

Store

```
procedure Store(var S: TStream);
```

Stores the collection and all its items on the stream *S*. *TCollection.Store* calls *TCollection.PutItem* for each item in the collection.

See also: *TCollection.PutItem*

TComboBox WObjects

TComboBox is an interface object that represents a corresponding combo box element in Windows. *TComboBox* objects are not normally used in dialog boxes (*TDialog*) or dialog windows (*TDlgWindow*), but are used when you want to display a standalone combo box as a child window in another window's client area. Combo box object inherit most of their functionality from *TListBox*.

There are three types of combo boxes: simple, drop down, and drop down list. These types are governed by Windows style constants: *cbs_Simple*, *cbs_DropDown*, and *cbs_DropDownList*. These constants are passed to the *Init* constructor, which in turn tells Windows which type of combo box element to create.

Field

TextLen

```
TextLen: Word;
```

Read only

TextLen contains the length of the character buffer in the edit portion of the combo box, which is also the number of bytes transferred by *Transfer*. *TextLen* is set by *Init*.

Methods

Init

```
constructor Init(AParent: PWindowsObject; AnID: Integer; X, Y, W, H:
  Integer; AStyle, ATextLen: Word);
```

Override: Sometimes

Constructs a combo box object with the passed parent window (*AParent*), control ID (*AnId*), position (*X*, *Y*) relative to the origin of the parent window's client area, width (*W*), and height (*H*), by calling *TListBox.Init*. Sets *TextLen* to *ATextLen*. Sets *Attr.Style* to (*Attr.Style* **and not** *lbs_Notify*) **or** *AStyle* **or** *cbs_Sort* **or** *ws_VScroll* **or** *ws_HScroll*.

See also: *TListBox.Init*, *cbs_* Combo box style constants

InitResource

```
constructor InitResource(AParent:PWindowsObject; ResourceID: Integer;
  ATextLen: Word);
```

Associates a *TComboBox* object with the resource indicated by *ResourceID*, with a maximum text length of *ATextLen* – 1.

Load

```
constructor Load(var S: TStream);
```

Constructs and loads a combo box from the stream *S* by first calling *TListBox.Load* and then reading the additional fields (*Style, TextLen*) introduced by *TComboBox*.

See also: *TListBox.Load*

GetClassName

```
function GetClassName: PChar; virtual;
```

Override: Never

Returns the name of *TComboBox's* window class, 'ComboBox'.

HideList

```
procedure HideList;
```

Forces the hiding of the drop down list of a drop down or drop down list combo box.

ShowList

```
procedure ShowList;
```

Forces the display of the drop down list of a drop down or drop down list combo box.

Store

```
procedure Store(var S: TStream);
```

Stores the combo box on the stream *S* by first calling *TListBox.Store* and then writing the additional fields (*Style, TextLen*) introduced by *TComboBox*.

See also: *TListBox.Store*

Transfer

function Transfer(DataPtr: Pointer; TransferFlag: Word); **virtual**;

Transfers data to or from the record pointed to by *DataPtr*. The record should be first a pointer to a string collection (*PStrCollection*) that holds the entries of the combo box's list. Then an array of characters holding the currently-selected entry is transferred. The transfer buffer would look something like this:

```
type
  TComboXferRec = record
    Strings: PCollection;
    Selection: array[0..TextLen-1] of Char;
  end;
```

where *TextLen* would be replaced by the value passed in the *Init* constructor.

If *TransferFlag* is *tf_GetData*, the combo box's data is transferred to the *DataPtr* record. If *TransferFlag* is *tf_SetData*, the data is transferred to the combo box from the record. In either of these cases, *Transfer* returns the size of the data transferred.

If *TransferFlag* is *tf_SizeData*, *Transfer* returns the size of the transfer data.

See also: *TListBox.Transfer*

TControl — WObjects

TControl is an abstract object type that, as an ancestor type, unifies all of the control object types, such as *TScrollBar* and *TButton*. It is also ancestor to *TMDIClient*, a specialized control for MDI-compliant applications. Control objects are generally not used in dialog boxes (*TDialog*) or dialog windows (*TDlgWindow*), but only as standalone controls in the client area of other windows.

Methods

Init

```
constructor Init(AParent: PWindowsObject; AnId: Integer; ATitle:
  PChar; X, Y, W, H: Integer);
```

Constructs a control object with the passed parent window (*AParent*), control ID (*AnId*); associated text (*ATitle*); position (*X*, *Y*), relative to the origin of the parent window's client area; width (*W*); and height (*H*). With these arguments, it fills the control's *Attr* field inherited from *TWindow*. As a default, it sets *Attr.Style* to *ws_Child* **or** *ws_Visible* **or** *ws_Group* **or** *ws_TabStop*, so that all control objects are visible child windows.

InitResource

```
constructor InitResource(AParent: PWindowsObject; ResourceID: Word);
```

Associates a control object with the control element in the resource specified by *ResourceID*. Calls *TWindow.InitResource* and *EnableTransfer* to enable the transfer mechanism.

See also: *TWindow.InitResource*, *TWindowsObject.EnableTransfer*

DefWndProc

```
procedure DefWndProc(var Msg: TMessage); virtual;
```

Override: Never

Calls the default window procedure for control objects. It stores the result from that procedure in the *Result* field of the passed *Msg* variable.

GetClassName

```
function GetClassName: PChar; virtual;
```

Override: Always

Abstract method to be overridden by descendant objects.

Register

```
function Register: Boolean; virtual;
```

Override: Never

Simply returns *True* to indicate that *TControl* descendants use pre-registered window classes.

WMPaint `procedure WMPaint(var Msg: TMessage); virtual wm_First + wm_Paint;`

Override: Seldom

Calls *DefWndProc* for standard repainting of control objects.

See also: *TControl.DefWndProc*

TDialog — WObjects

TDialog defines objects that serve as the modal and modeless dialog boxes used in Windows applications. A *TDialog* object has an associated dialog resource that describes the appearance and placement of its controls. This resource is specified in the call to *TDialog.Init*.

Dialog box objects can be associated with either modal or modeless dialog elements, through calls to its *Execute* or *Create* methods, respectively. Normally, however, dialog objects should be activated through the *TApplication* methods *ExecDialog* and *MakeWindow*, which check for low memory conditions before calling *Execute* or *Create*.

Note that the creation of a modal dialog disables the continued operations of its parent window.

Fields

Attr `Attr: TDialogAttr;`

Attr holds the dialog box's creation attributes. *TDialogAttr* is defined as follows:

```
TDialogAttr = record
  Name: PChar;
  Param: Longint;
end;
```

The *Name* field holds the name or ID of the dialog resource. The *Param* field holds a parameter that is passed to the dialog procedure when the dialog is created.

DialogProc `DialogProc: TFarProc;` **Read only**

DialogProc points to the procedure-instance address of the dialog function.

IsModal `IsModal: Boolean;` **Read only**

IsModal is *True* if the dialog is modal and *False* if it is modeless.

Methods

Init

```
constructor Init(AParent: PWindowsObject; AName: PChar);
```

Override: Sometimes

Constructs the dialog object by calling *TWindowsObject.Init*, passing the parent window, *AParent*. Then *TDialog.Init* sets the *Attr.Name* field to the string passed in *AName*. The string can be the symbolic name of the dialog resource, such as 'EMPLOYEEINFO' or an integer ID cast to type *PChar*, such as *PChar(120)*. *Init* also calls *DisableAutoCreate* so that dialogs are not automatically created and displayed along with their parent windows.

See also: *TWindowsObject.Init, TWindowsObject.DisableAutoCreate*

Load

```
constructor Load(var S: TStream);
```

Constructs and loads a dialog box from the stream *S* by first calling *TWindowsObject.Load* and then reading the additional fields (*Attr* and *IsModal*) introduced by *TDialog*.

See also: *TWindowsObject.Load*

Done

```
destructor Done; virtual;
```

Override: Sometimes

Disposes the dialog box object calls *TWindowsObject.Done*.

See also: *TWindowsObject.Done*

Cancel

```
procedure Cancel(var Msg: TMessage); virtual id_First + id_Cancel;
```

Override: Sometimes

Automatically responds to the pressing of a dialog's Cancel button by calling *EndDlg* with the value *id_Cancel*.

See also: *TDialog.EndDlg*

Create

```
function Create: Boolean; virtual;
```

Override: Never

Creates a modeless dialog object's corresponding dialog element. *TDialog.Create* returns *True* if successful. If unsuccessful, it calls *Error* with the error code *em_InvalidWindow*.

See also: *TDialog.Execute, TWindowsObject.Error*

DefWndProc

```
procedure DefWndProc(var Msg: TMessage); virtual;
```

Override: Never

Elicits Windows' default message processing by setting the result of the passed message equal to zero.

Destroy

```
procedure Destroy; virtual;
```

Cancels and destroys the dialog box by calling *EndDlg* and passing the constant *id_Cancel*.

See also: *TDialog.EndDlg*

EndDlg

```
procedure EndDlg(ARetValue: Integer); virtual;
```

Override: Never

Destroys modal and modeless dialog boxes. *ARetValue* is passed back as the return value from *TDialog.Execute* for modal dialog boxes.

See also: *TDialog.Execute, TDialog.Create*

EnterCancel

```
procedure EnterCancel(var Msg: TMessage); virtual cm_First + id_Cancel;
```

Override: Seldom

Automatically responds to the pressing of the *Enter* key in a dialog by calling *Cancel*. In this case, the Cancel button was the default or highlighted button.

See also: *TDialog.Cancel*

EnterOk

```
procedure EnterOk(var Msg: TMessage); virtual cm_First + id_OK;
```

Override: Seldom

Automatically responds to the pressing of the *Enter* key in a dialog by calling *Ok*. In this case, the OK button was the default or highlighted button.

See also: *TDialog.Ok*

Execute

```
function Execute: Integer; virtual;
```

Override: Never

Creates and displays a modal dialog object's corresponding dialog element. This method executes during the entire time the dialog box is on screen, until its *EndDlg* method is called. Before the method finishes, it resets *HWindow* to 0. *TDialog.Execute* returns the integer value returned by *TDialog.EndDlg* if successful. If unsuccessful, *Execute* calls *Error* with the error code *em_InvalidWindow*.

See also: *TDialog.EndDlg, TDialog.Create, TWindowsObject.Error*

GetItemHandle

```
function GetItemHandle(DlgItemID: Integer): HWnd;
```

Override: Never

Returns the handle to the dialog's control identified by the passed ID, *DlgItemID*.

Ok

```
procedure Ok(var Msg: TMessage); virtual id_First + id_OK;
```

Override: Sometimes

Automatically responds to the pressing of a dialog's OK button by calling *CanClose*, and *EndDlg* with the value *id_OK*. Also calls *TransferData(tf_GetData)* to transfer data from the controls to the transfer buffer.

See also: *TDialog.EndDlg*

SendDlgItemMsg

```
function SendDlgItemMsg(DlgItemID: Integer; AMsg, WParam: Word; LParam:
Longint): Longint;
```

Override: Never Sends a Windows control message, identified by *AMsg*, to the dialog's control identified by its passed ID, *DlgItemID*. *WParam* and *LParam* become parameters in the Windows message. *SendDlgItemMsg* returns the value returned by the control, or zero if the control ID is invalid.

SetName

```
procedure SetName(AName: PChar); virtual;
```

Sets the *Attr* field's *Name* field to the string passed in *AName*.

Store

```
procedure Store(var S: TStream);
```

Stores the dialog box on the stream *S* by first calling *TWindowsObject.Store* and then writing the additional fields (*Attr* and *IsModal*) introduced by *TDialog*.

See also: *TWindowsObject.Store*

WMInitDialog

```
procedure WMInitDialog(var Msg: TMessage); virtual wm_First +
wm_InitDialog;
```

Override: Never *TDialog.WMInitDialog* is automatically called just before the dialog is displayed. It calls *SetupWindow* to perform any initialization needed for the dialog or its controls.

See also: *TWindowsObject.SetupWindow*

TDlgWindow — WObjects

Dialog windows, defined by *TDlgWindow*, combine some characteristics of dialogs and of windows. Like a dialog, a dialog window has an associated dialog resource which describes the appearance and position of its controls. However, like a window, it has a window class that can specify icons and cursors. To create and display dialog windows, use the modeless dialog *Create* methods. Do not use the *Execute* method.

Methods

Init

```
constructor Init(AParent: PWindowsObject; AName: PChar);
```

Constructs a new *TDlgWindow* object by calling *TDialog.Init*. It also calls *EnableAutoCreate* so that, as a child window, it is automatically created and displayed along with its parent window.

See also: *TDialog.Init*, *TWindowsObject.EnableAutoCreate*

Cancel

```
procedure Cancel(var Msg: TMessage); virtual id_First + id_Cancel;
```

Override: Sometimes

Automatically responds to a click on the Cancel button by calling *CanClose* for window closing approval and then destroying the dialog window.

See also: *TDlgWindow.Ok*

Create

```
function Create: Boolean; virtual;
```

Override: Never

Registers the dialog window's class and calls *TDialog.Create*. *TDlgWindow.Create* returns *True* if successful.

See also: *TWindowsObject.Register, TDialog.Create*

GetWindowClass

```
procedure GetWindowClass(var AWndClass: TWndClass); virtual;
```

Override: Often

Defines the default window class record and passes it back in *AWndClass*. This window class specifies no menu and a standard icon and cursor. Redefine *GetWindowClass*, as well as *GetClassName* for your *TDlgWindow* descendants. However, make sure your *GetWindowClass* method calls *TDlgWindow.GetWindowClass* before modifying any *TWndClass* fields.

Ok

```
procedure Ok(var Msg: TMessage); virtual id_First + id_OK;
```

Override: Sometimes

Automatically responds to a click on the OK button by calling *CanClose* for window closing approval and then destroying the dialog window.

See also: *TDlgWindow.Cancel*

TDosStream — WObjects

TDosStream is a specialized *TStream* derivative implementing unbuffered DOS file streams. The constructor lets you create or open a DOS file by specifying its name and access mode: *stCreate, stOpenRead, stOpenWrite,* or *stOpen*. The one additional field of *TDosStream* is *Handle*, the traditional DOS file handle used to access an open file. Most applications will use the buffered derivative of *TDosStream* called *TBufStream*. *TDosStream* overrides all the abstract methods of *TStream* except for *TStream.Flush*.

Fields

Handle

```
Handle: Word
```

Read only

Handle is the DOS file handle used to access an open file stream.

Methods

Init `constructor Init(FileName: FNameStr; Mode: Word);`

Creates a DOS file stream with the given *FileName* and access mode. If successful, the *Handle* field is set with the DOS file handle. Failure is signaled by a call to *Error* with an argument of *stInitError*.

The *Mode* argument must be set to one of the values *stCreate*, *stOpenRead*, *stOpenWrite*, or *stOpen*. These constant values are explained in Chapter 6 under "stXXXX stream constants."

Done `destructor Done; virtual;`

Override: Never

Closes and disposes of the DOS file stream

See also: *TDosStream.Init*

GetPos `function GetPos: Longint; virtual;`

Override: Never

Returns the value of the calling stream's current position.

See also: *TDosStream.Seek*

GetSize `function GetSize: Longint; virtual;`

Override: Never

Returns the total size in bytes of the calling stream.

Read `procedure Read(var Buf; Count: Word); virtual;`

Override: Never

Reads *Count* bytes into the *Buf* buffer starting at the calling stream's current position.

See also: *TDosStream.Write*, *stReadError*

Seek `procedure Seek(Pos: Longint); virtual;`

Override: Never

Resets the current position to *Pos* bytes from the beginning of the stream.

See also: *TDosStream.GetPos*, *TDosStream.GetSize*

Truncate `procedure Truncate; virtual;`

Override: Never

Deletes all data on the calling stream from the current position to the end.

See also: *TDosStream.GetPos*, *TDosStream.Seek*

Write `procedure Write(var Buf; Count: Word); virtual;`

Writes *Count* bytes from the *Buf* buffer to the calling stream, starting at the current position.

See also: *TDosStream.Read*, *stWriteError*

TEdit WObjects

TEdit is an interface object that represents a corresponding edit control element in Windows. *TEdit* objects are not normally used in dialog boxes (*TDialog*) or dialog windows (*TDlgWindow*), but are used when you want to display a standalone edit control as a child window in another window's client area.

There are two styles edit control objects: single line and multiline. Multiline edit controls allow vertical scroll bars and editing in multiple lines. Most of *TEdit's* methods manage the edit control's text. *TEdit* also includes some command-based message response methods for automatically responding to cut, copy, paste, delete, clear and undo menu selections from the edit control's parent window. Two important methods inherited from *TEdit's* ancestor, *TStatic*, are *GetText* and *SetText*.

Field

IsMultiline

```
IsMultiline: Boolean;
```

Read only

IsMultiline is *True* in the case of a multiline edit control and *False* in the case of single line.

Methods

Init

```
constructor Init(AParent: PWindowsObject; AnId: Integer; ATitle:
  PChar; X, Y, W, H, ATextLen: Integer; Multiline: Boolean);
```

Override: Sometimes

Constructs an edit control object with a parent window (*AParent*), and fills its *Attr* fields with the passed control ID (*AnId*), initial text (*ATitle*), position (*X*, *Y*) relative to the origin of the parent window's client area, width (*W*), height (*H*), and text buffer length (*ATextLen*).

If *ATextLen* is zero, there is no explicit limit on the number of characters that can be entered. If *Multiline* is *True*, the edit control will be a multiline edit control with horizontal and vertical scroll bars. In that case, the *Attr.Style* field will include the Windows style constants *es_Multiline*, *es_AutoVScroll*, *es_AutoHScroll*, *es_Left*, *ws_VScroll*, and *ws_HScroll*. If *Multiline* is *False*, the edit control will have a single line of text and will have a border (*ws_Border*) and will be left justified (*es_Left*).

Load `constructor Load(var S: TStream);`

Constructs and loads an edit control from the stream *S* by first calling *TStatic.Load* and then reading the additional field (*IsMultiline*) introduced by *TEdit*.

See also: *TStatic.Load*

CanUndo `function CanUndo: Boolean; virtual;`

Override: Seldom

Returns *True* if it is possible to undo the last edit.

See also: *TEdit.Undo*

ClearModify `procedure ClearModify; virtual;`

Override: Seldom

Resets the changes flag for the edit control.

See also: *TEdit.IsModified*

CMEditClear `procedure CMEditClear(var Msg: TMessage); virtual cm_First + cm_EditClear;`

Override: Never

Automatically responds to a menu selection with a menu ID of *cm_EditClear* by calling the *Clear* method.

See also: *TEdit.Clear*

CMEditCopy `procedure CMEditCopy(var Msg: TMessage); virtual cm_First + cm_EditCopy;`

Override: Never

Automatically responds to a menu selection with a menu ID of *cm_EditCopy* by calling the *Copy* method.

See also: *TEdit.Copy*

CMEditCut `procedure CMEditCut(var Msg: TMessage); virtual cm_First + cm_EditCut;`

Override: Never

Automatically responds to a menu selection with a menu ID of *cm_EditCut* by calling the *Cut* method.

See also: *TEdit.Cut*

CMEditDelete `procedure CMEditDelete(var Msg: TMessage); virtual cm_First + cm_EditDelete;`

Override: Never

Automatically responds to a menu selection with a menu ID of *cm_EditDelete* by calling the *DeleteSelection* method.

See also: *TEdit.DeleteSelection*

CMEditPaste `procedure CMEditPaste(var Msg: TMessage); virtual cm_First + cm_EditPaste;`

Override: Never Automatically responds to a menu selection with a menu ID of *cm_EditPaste* by calling the *Paste* method.

See also: *TEdit.Paste*

CMEditUndo

```
procedure CMEditUndo(var Msg: TMessage); virtual cm_First + cm_EditUndo;
```

Override: Never Automatically responds to a menu selection with a menu ID of *cm_EditUndo* by calling the *Undo* method.

See also: *TEdit.Undo*

Copy

```
procedure Copy; virtual;
```

Override: Seldom Copies the currently selected text into the clipboard.

See also: *TEdit.CMEditCopy*

Cut

```
procedure Cut; virtual;
```

Override: Seldom Cuts (copies and deletes) the currently selected text into the clipboard.

See also: *TEdit.CMEditCut*

DeleteLine

```
function DeleteLine(LineNumber: Integer): Boolean; virtual;
```

Override: Sometimes Deletes the text in the line specified by *LineNumber* in a multiline edit control. *DeleteLine* does not delete the line break and affects no other lines. The method returns *True* if successful.

DeleteSelection

```
function DeleteSelection: Boolean; virtual;
```

Override: Seldom Clears the currently selected text, and returns *False* if no text is selected.

See also: *TEdit.CMEditDelete*

DeleteSubText

```
function DeleteSubText(StartPos, EndPos: Integer): Boolean; virtual;
```

Override: Seldom Deletes the text between the starting and ending positions specified by *StartPos* and *EndPos*. The first character is in position zero, and the position numbers continue sequentially throughout all lines in a multiline edit control. Line breaks count as two characters. *DeleteSubText* returns *True* if successful.

GetClassName

```
function GetClassName: PChar; virtual;
```

Override: Never Returns the name of *TEdit's* window class, 'Edit'.

GetLine

```
function GetLine(ATextString: PChar; StrSize, LineNumber: Integer):
Boolean; virtual;
```

Override: Seldom Retrieves a multiline edit control's text from the line specified by *LineNumber* and returns it in *ATextString*. *StrSize* indicates how many

characters to retrieve. *GetLine* returns false if it is unable to retrieve the text or if it is too long.

See also: *TStatic.GetText, TEdit.GetNumLines, TEdit.LineLength*

GetLineIndex

```
function GetLineIndex(LineNumber: Integer): Integer; virtual;
```

Override: Seldom

Returns, from a multiline edit control, the number of characters that appear before the line number specified by *LineNumber*. Line breaks count as two characters. If the specified line does not exist, *GetLineIndex* returns the total number of characters in the edit control.

GetLineFromPos

```
function GetLineFromPos(CharPos: Integer): Integer; virtual;
```

Override: Seldom

Returns, from a multiline edit control, the line number on which the character position specified by *CharPos* occurs. The position of the first character is zero and the numbers continue sequentially throughout all of the lines. Line breaks count as two characters.

GetLineLength

```
function GetLineLength(LineNumber: Integer): Integer; virtual;
```

Override: Seldom

Returns, from a multiline edit control, the number of characters in the line specified by *LineNumber*. *GetLineLength* should be called before calling *GetLine*.

See also: *TEdit.GetLine*

GetNumLines

```
function GetNumLines: Integer; virtual;
```

Override: Seldom

Returns the number of lines that have been entered in a multiline edit control. *GetNumLines* should be called before calling *GetLine*.

See also: *TEdit.GetLine*

GetSelection

```
procedure GetSelection(var StartPos, EndPos: Integer); virtual;
```

Override: Seldom

Retrieves the starting and ending positions of the currently selected text and returns them in the *StartPos* and *EndPos* arguments. The first character is at position zero. In a multiline edit control, the positions are counted sequentially throughout all lines, and line breaks count for two characters. When using *GetSelection* in conjunction with *GetSubText*, you can get the currently selected text.

See also: *TEdit.GetSubText*

GetSubText

```
procedure GetSubText(ATextString: PChar; StartPos, EndPos: Integer);
  virtual;
```

Override: Seldom

Retrieves, in *ATextString*, the text in an edit control from indexes *StartPos* to *EndPos*. When specifying the range, the first character is at index zero.

In a multiline edit control, the characters are counted sequentially throughout all of the lines, and line breaks are counted as two characters.

See also: *TEdit.GetSelection*

Insert

```
procedure Insert(ATextString: PChar); virtual;
```

Override: Seldom

Inserts the text passed in *ATextString* into the edit control at the current text insertion point, and replaces any currently selected text. *Insert* is similar to *Paste*, but does not affect the clipboard.

See also: *TEdit.Paste*

IsModified

```
function IsModified: Boolean; virtual;
```

Override: Seldom

Returns *True* if the user has changed the text in the edit control.

See also: *TEdit.ClearModify*

Paste

```
procedure Paste; virtual;
```

Override: Seldom

Inserts text from the clipboard into the edit control at the current insertion point.

See also: *TEdit.CMEditPaste*

Scroll

```
procedure Scroll(HorizontalUnit, VerticalUnit: Integer); virtual;
```

Override: Seldom

Scrolls a multiline edit control horizontally and vertically by the numbers of characters specified in *HorizontalUnit* and *VerticalUnit*. Positive values result in a scroll to the right or down, and negative values scroll to the left or up.

SetSelection

```
function SetSelection(StartPos, EndPos: Integer): Boolean; virtual;
```

Override: Seldom

Forces the selection of the text between the positions specified by *StartPos* and *EndPos*, but not including the character at *EndPos*. The first character is in position zero, and the position numbers continue sequentially throughout all lines in a multiline edit control. Line breaks count as two characters.

SetupWindow

```
procedure SetupWindow; virtual;
```

Limits the number of characters that can be entered into the edit control, by calling *TStatic.SetupWindow*, then if the *TextLen* field is non-zero, the value *TextLen* – 1 is sent to Windows through an *em_LimitText* message.

See also: *TStatic.SetupWindow*, *em_LimitText* message

Store `procedure Store(var S: TStream);`

Stores the edit control on the stream *S* by first calling *TStatic.Store* and then writing the additional field (*IsMultiline*) introduced by *TEdit*.

See also: *TStatic.Store*

Transfer `function Transfer(DataPtr: Pointer; TransferFlag: Word): Word; virtual;`

Override: Sometimes

Transfers *TextLen* characters of the current text of the edit control to or from the memory location pointed to by *DataPtr*. If *TransferFlag* is *tf_GetData*, the text is transferred to the memory location. If *TransferFlag* is *tf_SetData*, the edit control's text is set to the text at the memory location. *Transfer* returns *TextLen*, the number of bytes stored in or retrieved from the memory location. If *TransferFlag* is *tf_SizeData*, *Transfer* returns the size of the transfer data.

Undo `procedure Undo; virtual;`

Override: Seldom

Undoes the last edit.

See also: *TEdit.CanUndo*, *TEdit.CMEditUndo*

TEmsStream — WObjects

TEmsStream is a specialized *TStream* derivative for implementing streams in EMS memory. The additional fields provide an EMS handle, a page count, stream size, and current position. *TEmsStream* overrides the six abstract methods of *TStream* as well as providing a specialized constructor and destructor.

When debugging a program using EMS streams, the IDE cannot recover EMS memory allocated by your program if your program terminates prematurely or if you forget to call the *Done* destructor for an EMS stream. Only the *Done* method (or rebooting) can release the EMS pages owned by the stream.

Fields

Handle `Handle: Word;` **Read only**

The EMS handle for the stream.

PageCount `PageCount: Word;` **Read only**

The number of allocated pages for the stream, with 16K per page.

Position `Position: Longint;` **Read only**

The current position within the stream. The first position is 0.

Size `Size: Longint;` **Read only**

The size of the stream in bytes.

Methods

Init `constructor Init(MinSize, MaxSize: Longint);`

Creates an EMS stream with the given minimum and maximum sizes in bytes. Calls *TStream.Init* then sets *Handle*, *Size* and *PageCount*. Calls *Error* with an argument of *stInitError* if initialization fails.

See also: *TEmsStream.Done*

Done `destructor Done; virtual;`

Override: Never

Disposes of the EMS stream and releases EMS pages used.

See also: *TEmsStream.Init*

GetPos `function GetPos: Longint; virtual;`

Override: Never

Returns the value of the calling stream's current position.

See also: *TEmsStream.Seek*

GetSize `function GetSize: Longint; virtual;`

Override: Never

Returns the total size of the calling stream.

Read `procedure Read(var Buf; Count: Word); virtual;`

Override: Never

Reads *Count* bytes into the *Buf* buffer starting at the calling stream's current position.

See also: *TEmsStream.Write, stReadError*

Seek `procedure Seek(Pos: Longint); virtual;`

Override: Never

Resets the current position to *Pos* bytes from the start of the calling stream.

See also: *TEmsStream.GetPos, TEmsStream.GetSize*

Truncate `procedure Truncate; virtual;`

Override: Never Deletes all data on the calling stream from the current position to the end. The current position is set to the new end of the stream.

See also: *TEmsStream.GetPos, TEmsStream.Seek*

Write `procedure Write(var Buf; Count: Word); virtual;`

Override: Never Writes *Count* bytes from the *Buf* buffer to the calling stream, starting at the current position.

See also: *TStream.Read, TEmsStream.GetPos, TEmsStream.Seek*

TGroupBox — WObjects

TGroupBox is an interface object that represents a corresponding group box element in Windows. *TGroupBox* objects are not normally used in dialog boxes (*TDialog*) or dialog windows (*TDlgWindow*), but are used when you want to display a standalone group box as a child window in another window's client area.

While group boxes don't serve an active purpose on the screen, they visually unify a group of selection boxes (check boxes and radio buttons). Behind the scenes, however, they perform the important role in managing the states of their selection boxes. For example you may want to respond to the user checking one box by unchecking all of the others.

Field

NotifyParent `NotifyParent: Boolean;` **Read/write**

Flag which indicates whether or not parent is to be notified when the state of the group box's selection boxes has changed.

Methods

Init

```
constructor Init(AParent: PWindowsObject; AnID: Integer; AText: PChar;
  X, Y, W, H: Integer);
```

Override: Sometimes Constructs a group box object with the passed parent window (*AParent*), control ID (*AnId*), associated text (*AText*), position (*X, Y*), relative to the origin of the parent window's client area; width (*W*), and height (*H*). Calls *TControl.Init* and then adds the Windows style *bs_GroupBox* to, and removes the style *ws_TabStop* from, the group box's *Attr.Style* field.

NotifyParent is set to *True*; by default, the group box's parent is notified when the state of its selection boxes changes.

See also: *TControl.Init*

InitResource

```
constructor InitResource(AParent: PWindowsObject; ResourceID: Word);
```

Subclasses the group box by constructing an ObjectWindows object to correspond to a group box element created by a dialog resource definition. Calls *TControl.InitResource* and *DisableTransfer* to exclude group boxes from the transfer mechanism, since they have no data to be transferred.

See also: *TControl.InitResource, TWindowsObject.DisableTransfer*

Load

```
constructor Load(var S: TStream);
```

Constructs and loads a group box from the stream *S* by first calling *TControl.Load* and then reading the additional field (*NotifyParent*) introduced by *TGroupBox*.

See also: *TControl.Load*

GetClassName

```
function GetClassName: PChar; virtual;
```

Override: Sometimes

Returns the name of *TGroupBox*'s window class, 'Button'.

SelectionChanged

```
procedure SelectionChanged(ControlId: Integer); virtual;
```

Override: Sometimes

If *NotifyParent* is *True*, notifies the parent window of the group box that one of its selections has changed by sending it a child-ID-based message. This method could be overridden to allow the group box to respond to its selections.

Store

```
procedure Store(var S: TStream);
```

Stores the group box on the stream *S* by first calling *TControl.Store* and then writing the additional field (*NotifyParent*) introduced by *TGroupBox*.

See also: *TControl.Store*

TListBox WObjects

TListBox is an interface object that represents a corresponding list box element in Windows. *TListBox* objects are not normally used in dialog boxes (*TDialog*) or dialog windows (*TDlgWindow*), but are used when you want to display a standalone list box as a child window in another window's client area. *TListBox*'s methods also serve instances of its descendant, *TComboBox*.

Methods

Init

```
constructor Init(AParent: PWindowsObject; AnId: Integer; X,Y,W,H:
Integer);
```

Override: Sometimes

Constructs a list box object with the passed parent window (*AParent*) control ID (*AnId*), position (*X*, *Y*), relative to the origin of the parent window's client area, width (*W*), and height (*H*). Calls *TControl.Init* and adds to the list box object's *Attr.Style* field the Windows style constant *lbs_Standard*, which provides the scroll bar with:

- A border (*ws_Border*)
- A vertical scroll bar (*ws_VScroll*)
- Automatic alphabetic sorting of list items (*lbs_Sort*)
- Parent window notification upon selection (*lbs_Notify*)

These styles can be overridden in a descendant class, or in the *Init* constructor of the list box's parent window object.

AddString

```
function AddString(AString: PChar): Integer; virtual;
```

Override: Sometimes

Adds *AString* as a list item in the list box object, and returns the item's position index (starting at zero) or a negative value in the case of an error. The list items are automatically sorted unless the style *lbs_Sort* is removed from the list box object's *Attr.Style* field before creation.

ClearList

```
procedure ClearList; virtual;
```

Override: Sometimes

Removes all of the list items from the list box.

DeleteString

```
function DeleteString(Index: Integer): Integer; virtual;
```

Override: Sometimes

Removes the list item at the position index (starting at zero) passed in *Index*. *DeleteString* returns the number of remaining list items, or a negative value in the case of an error.

GetClassName

```
function GetClassName: PChar; virtual;
```

Override: Never

Returns the name of *TListBox*'s window class, 'ListBox'.

GetCount

```
function GetCount: Integer; virtual;
```

Override: Seldom

Returns the number of list items in the list box, or a negative value in the case of an error.

GetMsgID

function GetMsgID(AMsg: TMsgName): Word; **virtual**;

Translates list box messages for use by *TComboBox* objects.

GetSelIndex

```
function GetSelIndex: Integer; virtual;
```

Override: Seldom

Returns the position index (starting at zero) of the currently selected list item, or a negative value in the case that no item is selected.

GetSelString

```
function GetSelString(AString: PChar; MaxChars: Integer): Integer;
virtual;
```

Override: Seldom

Retrieves in *AString* the currently selected list item, as long as it is no longer than *MaxChars*. *GetSelString* returns the string length or a negative value in the case of an error.

GetString

```
function GetString(AString: PChar; Index: Integer): Integer; virtual;
```

Override: Seldom

Retrieves in *AString* the list item at the position index (starting at zero) passed in *Index*, and returns the string length or a negative value in the case of an error.

GetStringLen

```
function GetStringLen(Index: Integer): Integer; virtual;
```

Override: Seldom

Returns the string length of the list item at the position index passed in *Index*, or a negative value in the case of an error.

InsertString

```
function InsertString(AString: PChar; Index: Integer): Integer; virtual;
```

Override: Sometimes

Inserts *AString* as a list item in the list box at the position index passed in *Index*, and returns the item's actual position (starting at zero) or a negative value in the case of an error. The list box items are not resorted. If *Index* is –1, the string is appended to the end of the list.

SetSelIndex

```
function SetSelIndex(Index: Integer): Integer; virtual;
```

Override: Seldom

Forces the selection of the list item at the position index (starting at zero) passed in *Index*. If *Index* is –1, the list box is cleared of any selection. *SetSelIndex* returns a negative number in the case of an error.

SetSelString

```
function SetSelString(AString: PChar; AIndex: Integer): Integer;
virtual;
```

Override: Seldom

Forces the selection of the first list item matching the text passed in *AString* that appears beyond the position index (starting at zero) passed in *AIndex*. If *AIndex* is –1, the entire list is searched from the beginning. *SetSelString* returns the position index of the newly selected item, or a negative value in the case of an error.

Transfer

```
function Transfer(DataPtr: Pointer; TransferFlag: Word): Word;
virtual;
```

Override: Sometimes

Transfers the list of entries and selected item(s) to or from the transfer record pointed to by *DataPtr*. If *TransferFlag* is *tf_GetData*, the list box's

data is transferred to the memory location. If *TransferFlag* is *tf_SetData*, the list box is loaded with the data from the memory location. *Transfer* returns the number of bytes transferred. If *TransferFlag* is *tf_SizeData*, *Transfer* returns the size of the transfer data.

The nature of the record varies somewhat, depending on whether the box allows multiple items to be selected. The first item transferred is always a pointer to a collection of strings which are the list box's entries. For single-selection list boxes, that is followed by an integer index to the selected item. For multiple-selection list boxes, the collection is followed by a pointer to a *TMultiSelRec* record, which contains an array of integers, each indicating a selected item.

Typical transfer records for list boxes look like this:

```
type
  TListBoxXferRec = record                          { single selection }
    Strings: PStrCollection;
    Selection: Integer;
  end;

  TMultiListXferRec = record                        { multiple selection }
    Strings: PStrCollection;
    Selections: PMultiSelRec;
  end;
```

PMultiSelRec is a record defined in the *WObjects* unit. The record must be allocated using the *AllocMultiSel* function, and freed by *FreeMultiSel* after the transfer. A **nil** pointer indicates that no entries are selected.

TMDIClient — WObjects

Multiple Document Interface (MDI) client windows, represented by TMDIClient, are controls that manage the MDI child windows of an MDI-compliant application. TMDIClient's methods are concerned with managing MDI child windows.

Field

ClientAttr `ClientAttr: TClientCreateStruct;`

ClientAttr holds a record of the MDI client window's attributes. *TClientCreateStruct* is defined as:

```
type
  PClientCreateStruct = ^TClientCreateStruct;
  TClientCreateStruct = record
    hWindowMenu: THandle;
    idFirstChild: Word;
  end;
```

Methods

Init

```
constructor Init(AParent: PMDIWindow);
```

Override: Seldom

Constructs the MDI client window object with *AParent* as the parent window. *Init* sets the fields of *ClientAttr*. Calls *TControl.Init* and adds the Windows style *ws_ClipChildren* to the window object's *Attr.Style* field. In addition, *Init* removes the client window from its parent's child window list, so it is not treated as other child windows, such as list boxes and buttons.

See also: *TControl.Init*

Load

```
constructor Load(var S: TStream);
```

Constructs and loads an MDI client window from the stream *S* by first calling *TControl.Load* and then reading the additional field (*ClientAttr*) introduced by *TMDIClient*.

See also: *TControl.Load*

ArrangeIcons

```
procedure ArrangeIcons; virtual;
```

Override: Seldom

Arranges the minimized MDI child windows into a neat row at the bottom of the MDI client window.

CascadeChildren

```
procedure CascadeChildren; virtual;
```

Override: Seldom

Sizes and arranges all of the non-minimized MDI child windows within the MDI client window so as to overlap and display the title bar of each one.

GetClassName

```
function GetClassName: PChar; virtual;
```

Override: Never

Returns *TMDIClient*'s window class name, 'MDIClient'.

Store

```
procedure Store(var S: TStream);
```

Stores the MDI client window on the stream *S* by first calling *TControl.Store* and then writing the additional field (*ClientAttr*) introduced by *TMDIClient*.

See also: *TControl.Store*

TileChildren `procedure TileChildren; virtual;`

Override: Seldom

Sizes and arranges all of the non-minimized MDI child windows within the MDI client window so as to take up all the available space without overlapping.

TMDIWindow — WObjects

Multiple Document Interface (MDI) frame windows, represented by *TMDIWindow*, are overlapped windows that serve as the main window of MDI-compliant applications. One major feature of *TMDIWindow* objects is that they own a *TMDIClient* object and store it in the *ClientWnd* field. Another feature is the child window menu that offers options for manipulating the application's MDI child windows. This window is automatically modified to reflect all displayed MDI child windows.

Fields

ChildMenuPos `ChildMenuPos: Integer;` **Read/write**

ChildMenuPos specifies an index identifying the position of the child window management menu. The index counts only top-level menu items and the top-left item is at position zero.

ClientWnd `ClientWnd: PMDIClient;` **Read only**

ClientWnd points to the MDI frame window's MDI client window, an object instance of *TMDIClient*.

Methods

Init `constructor Init(ATitle: PChar; AMenu: HMenu);`

Override: Often

Constructs an MDI frame window object using the caption passed in *ATitle* and the menu passed in *AMenu*. MDI frame windows are required to have menus. An MDI frame window has no parent window, and must be the main window of the application. As a default, *Init* sets *ChildMenuPos* to zero, indicating that the child window menu is the top-left menu. To modify *ChildMenuPos*, override *TMDIWindow.Init* in your descendant types. For example:

```
constructor MyMDIWindow.Init(ATitle: PChar; AMenu: HMenu);
begin
  TMDIWindow.Init(ATitle, AMenu);
```

```
    ChildMenuPos := 3;
  end;
```

See also: *TMDIWindow.InitClientWindow*

Load

```
constructor Load(var S: TStream);
```

Constructs and loads an MDI frame window from the stream *S* by first calling *TWindow.Load* and then getting and reading the additional fields (*ClientWnd* and *ChildMenuPos*) introduced by *TMDIWindow*.

See also: *TWindow.Load*

Done

```
destructor Done; virtual;
```

Override: Sometimes

Disposes the MDI client window object stored in *ClientWnd* before calling *TWindow.Done* to dispose the MDI frame window object.

See also: *TWindow.Done*

ArrangeIcons

```
procedure ArrangeIcons;
```

Override: Seldom

Arranges the minimized MDI child windows into a neat row at the bottom of the MDI client window. Calls *ClientWnd^.ArrangeIcons*.

See also: *TMDIClient.ArrangeIcons*

CascadeChildren

```
procedure CascadeChildren;
```

Override: Seldom

Sizes and arranges all of the non-minimized MDI child windows within the MDI client window so as to overlap and display the title bar of each one. Calls *ClientWnd^.CascadeChildren*.

See also: *TMDIClient.CascadeChildren*

CloseChildren

```
procedure CloseChildren;
```

Override: Seldom

Destroys all of the created MDI child windows for which *CanClose* returns *True*.

See also: *TMDIClient.CloseChildren*

CMArrangeIcons

```
procedure CMArrangeIcons(var Msg: TMessage); virtual cm_First +
cm_ArrangeIcons;
```

Override: Seldom

Responds to a menu selection with an ID of *cm_ArrangeIcons* by calling *ArrangeIcons*.

See also: *TMDIWindow.ArrangeIcons*

CMCascadeChildren

```
procedure CMCascadeChildren(var Msg: TMessage); virtual cm_First +
cm_CascadeChildren;
```

Override: Seldom

Responds to a menu selection with an ID of *cm_CascadeChildren* by calling *CascadeChildren*.

See also: *TMDIWindow.CascadeChildren*

CMCloseChildren

```
procedure CMCloseChildren(var Msg: TMessage); virtual cm_First +
cm_CloseChildren;
```

Override: Seldom

Responds to a menu selection with an ID of *cm_CloseChildren* by calling *CloseChildren*.

See also: *TMDIWindow.CloseChildren*

CMCreateChild

```
procedure CMCreateChild(var Msg: TMessage); virtual cm_First +
cm_CreateChild;
```

Override: Never

Responds to a menu selection with a menu ID of *cm_CreateChild* by calling *CreateChild* to produce a new child window.

See also: *TMDIWindow.CreateChild*

CMTileChildren

```
procedure CMTileChildren(var Msg: TMessage); virtual cm_First +
cm_TileChildren;
```

Override: Seldom

Responds to a menu selection with an ID of *cm_TileChildren* by calling *TileChildren*.

See also: *TMDIWindow.TileChildren*

CreateChild

```
function CreateChild: PWindowsObject; virtual;
```

Constructs and creates a new MDI child window by calling *InitChild* and *MakeWindow*. You need not override *CreateChild* to accommodate descendant MDI child window types, as is true with *TMDIWindow.InitChild*. *CreateChild* returns a pointer to the new MDI child window.

See also: *TMDIWindow.InitChild, TApplication.MakeWindow*

DefWndProc

```
procedure DefWndProc(var Msg: TMessage); virtual;
```

Overrides *TWindow*'s default Windows message processing by calling the Windows function *DefFrameProc* rather than *DefWindowProc*.

See also: *TWindow.DefWndProc*

GetClassName

```
function GetClassName: PChar; virtual;
```

Override: Sometimes

Returns the name of *TMDIWindow*'s window class name, 'TurboMDIWindow'.

GetClient

```
function GetClient: PMDIClient; virtual;
```

Override: Never

Returns a pointer to the MDI client window stored in *ClientWnd*.

GetWindowClass

```
procedure GetWindowClass(var AWndClass: TWndClass); virtual;
```

Override: Sometimes

Modifies the default window class record and passes it back in *AWndClass*. *GetWindowClass* sets the style field to zero to remove the styles set by *TWindow.GetWindowClass*.

See also: *TWindow.GetWindowClass*

InitChild

```
function InitChild: PWindowsObject; virtual;
```

Override: Often

Constructs an MDI child window object (*TWindow*) with a caption of 'MDI Child' and returns a pointer to it. If you define an MDI child window type descending from *TWindow*, override *TMDIWindow.InitChild* to construct a window of your new MDI child window type. For example:

```
function MyMDIWindow.InitChild: PWindowsObject;
begin
  InitChild := New(PMyMDIChild, Init(@Self, 'Untitled Window'));
end;
```

See also: *TMDIWindow.CreateChild*

InitClientWindow

```
procedure InitClientWindow; virtual;
```

Override: Sometimes

Constructs the MDI client window as a *TMDIClient* object and stores it in *ClientWnd*.

SetupWindow

```
procedure SetupWindow; virtual;
```

Override: Often

Constructs the MDI client window (*ClientWnd*) object's corresponding window element by calling *InitClientWindow*, and creates it by calling *MakeWindow*. If you override *TMDIWindow.SetupWindow* in a descendant type, be sure to call *TMDIWindow.SetupWindow* explicitly.

See also: *TMDIWindow.InitClientWindow*, *TApplication.MakeWindow*

Store

```
procedure Store(var S: TStream);
```

Stores the MDI frame window on the stream *S* by first calling *TWindow.Store* and then putting and writing the additional fields (*ClientWnd* and *ChildMenuPos*) introduced by *TMDIWindow*.

See also: *TWindow.Store*

TileChildren

```
procedure TileChildren;
```

Override: Seldom

Sizes and arranges all of the non-minimized MDI child windows within the MDI client window so as to take up all the available space without overlapping. Calls *ClientWnd^.TileChildren*.

See also: *TMDIClient.TileChildren*

TObject WObjects

TObject is the starting point of the ObjectWindows object hierarchy. As the base object, it has no parents but many descendants. All of the ObjectWindows standard objects are ultimately derived from *TObject*. Any object that uses the ObjectWindows streams facilities *must* trace its ancestry back to *TObject*.

Methods

Init `constructor Init;`

Allocates space on the heap for the object. Called by all derived objects' constructors.

Free `procedure Free;`

Disposes of the object and calls the *Done* destructor.

Done `destructor Done; virtual;`

Performs the necessary cleanup and disposal for dynamic objects.

TRadioButton WObjects

TRadioButton is an interface object that represents a corresponding radio button element in Windows. *TRadioButton* objects are not normally used in dialog boxes (*TDialog*) or dialog windows (*TDlgWindow*), but are used when you want to display a standalone radio button as a child window in another window's client area.

Radio buttons have two states: checked and unchecked. *TRadioButton* inherits its state management methods from its ancestor, *TCheckBox*. Optionally, a check box can be part of a group (*TGroupBox*) which visually and conceptually groups its controls.

Methods

Init

```
constructor Init(AParent: PWindowsObject; AnID: Integer; ATitle: PChar;
X,Y,W,H: Integer; AGroup: PGroupBox);
```

Override: Sometimes

Constructs a radio button object with the passed parent window (*AParent*), control ID (*AnId*), associated text (*ATitle*), position (*X*, *Y*) relative to the origin of the parent window's client area; width (*W*), height (*H*), and associated group box (*AGroup*). *TRadioButton.Init* sets the check box's *Attr.Style* field to *ws_Child* **or** *ws_Visible* **or** *bs_RadioButton*.

TScrollBar — WObjects

TScrollBar objects represent standalone scroll bar controls, but not the scroll bars that are attached to windows. Scroll bars can be vertical (scrolls up and down) or horizontal (scrolls right and left). Most of *TScrollBar*'s Methods are concerned with managing the scroll bar's thumb position and range.

One special feature of the type *TScrollBar* is the set of methods that automatically respond to the Windows scroll bar messages, *wm_HScroll* and *wm_VScroll*. The methods, such as *SBLineUp* and *SBPageDown*, are defined as notify-based methods. These methods automatically adjust the scroll bar's thumb position.

TScrollBar objects should never be placed in windows that have either the *ws_HScroll* or *ws_VScroll* styles in their attributes.

Fields

IsHorizontal

```
IsHorizontal: Boolean;
```

Read/write

This field is *True* if the scroll bar is horizontal and *False* if it is vertical.

LineMagnitude

```
LineMagnitude: Integer;
```

Read/write

LineMagnitude is the number of range units to scroll the scroll bar when the user requests a small movement by clicking on the scroll bar's arrows. *Init* sets *LineMagnitude* to 1 by default. *TScrollBar.InitWindow* sets the scroll range from zero to 100 by default.

See also: *TScrollBar.InitWindow*

PageMagnitude

```
PageMagnitude: Integer;
```

Read/write

PageMagnitude is the number of range units to scroll the scroll bar when the user requests a large movement by clicking in the scroll bar's scrolling

area. *TScrollBar.Init* sets *PageMagnitude* to 10 by default. (The scroll range is set to from zero to 100 by default.)

Methods

Init

```
constructor Init(AParent: PWindowsObject; AnID: Integer; X, Y, W, H:
  Integer; IsHScrollBar: Boolean);
```

Constructs and initializes a *TScrollBar* object with the given parent window (*AParent*), *AnID* as a control ID, a position of (*X*, *Y*), a width of *W* and a height of *H*. The scroll bar is horizontal (style *sbs_Horz*) if *IsHScrollBar* is *True* and vertical (style *sbs_Vert*) if it is *False*. If the supplied height for a horizontal scroll bar or the supplied width for a vertical scroll bar is zero, a standard value is used. *LineMagnitude* is initialized to 1 and *PageMagnitude* to 10.

Load

```
constructor Load(var S: TStream);
```

Constructs and loads a scroll bar control from the stream *S* by first calling *TControl.Load* and then reading the additional field (*IsHorizontal*, *LineMagnitude* and *PageMagnitude*) introduced by *TScrollBar*.

See also: *TControl.Load*

DeltaPos

```
function DeltaPos(Delta: Integer): Integer; virtual;
```

Override: Seldom

Changes the scroll bar's thumb position by the value passed in *Delta*. (Calls *SetPosition*) A negative value moves the thumb up or left. The new thumb position is returned.

GetClassName

```
function GetClassName: PChar; virtual;
```

Override: Never

Returns the name of *TScrollBar*'s window class, 'Scrollbar'.

GetPosition

```
function GetPosition: Integer; virtual;
```

Override: Seldom

Returns the scroll bar's current thumb position.

See also: *TScrollBar.SetPosition*

GetRange

```
procedure GetRange(var LoVal, HiVal: Integer); virtual;
```

Override: Seldom

Retrieves the allowed range of scroll bar thumb positions in *LoVal* and *HiVal*.

See also: *TScrollBar.SetRange*

SBBottom

```
procedure SBBottom(var Msg: TMessage); virtual nf_First + sb_Bottom;
```

Override: Seldom Sets the thumb position to the highest allowable value in response to the user's request. This method is called in response to a scroll-based message carrying the code *sb_Bottom*. (Calls *SetPosition*)

SBLineDown `procedure SBLineDown(var Msg: TMessage); virtual nf_First + sb_LineDown;`

Override: Seldom Moves the thumb position down or right by *LineMagnitude*. *SBLineDown* is called automatically in response to a scroll-based message carrying the code *sb_LineDown*. (Calls *SetPosition*)

SBLineUp `procedure SBLineUp(var Msg: TMessage); virtual nf_First + sb_LineUp;`

Override: Seldom Moves the thumb position up or left by *LineMagnitude*. *SBLineUp* is called automatically in response to a scroll-based message carrying the code *sb_LineUp*. (Calls *SetPosition*) Sent when the user clicks on the scrollbar's "up" arrow, etc.

SBPageDown `procedure SBPageDown(var Msg: TMessage); virtual nf_First + sb_PageDown;`

Override: Seldom Moves the thumb position down or right by *PageMagnitude*. *SBPageDown* is called automatically in response to a scroll-based message carrying the code *sb_PageDown*. (Calls *SetPosition*)

SBPageUp `procedure SBPageUp(var Msg: TMessage); virtual nf_First + sb_PageUp;`

Override: Seldom Moves the thumb position up or left by *PageMagnitude*. *SBPageUp* is called automatically in response to a scroll-based message carrying the code *sb_PageUp*. (Calls *SetPosition*)

SBThumbPosition `procedure SBThumbPosition(var Msg: TMessage); virtual nf_First + sb_ThumbPosition;`

Override: Seldom Changes the thumb position to position chosen by the user and passed in a scroll-based message carrying the code *sb_ThumbPosition*. (Calls *SetPosition*)

SBThumbTrack `procedure SBThumbTrack(var Msg: TMessage); virtual nf_First + sb_ThumbTrack;`

Override: Sometimes Changes the thumb position as the user drags the thumb. (Calls *SetPosition*) This method is called in response to a scroll-based message carrying the code *sb_ThumbTrack*.

SBTop `procedure SBTop(var Msg: TMessage); virtual nf_First + sb_Top;`

Override: Seldom Sets the thumb position to the lowest allowable value in response to the user's request. This method is called in response to a scroll-based message carrying the code *sb_Top*. (Calls *SetPosition*)

SetPosition `procedure SetPosition(ThumbPos: Integer); virtual;`

Override: Sometimes

Sets the scroll bar's thumb position to *ThumbPos*. If *ThumbPos* is higher than the allowable range, the thumb position is set to the highest allowable value. If *ThumbPos* is lower than the allowable range, the thumb is set to the lowest possible value.

See also: *TScrollBar.GetPosition*

SetRange

```
procedure SetRange(LoVal, HiVal: Integer); virtual;
```

Override: Seldom

Sets the scroll bar's allowable range of thumb positions from *LoVal* to *HiVal*.

See also: *TScrollBar.GetRange*

SetupWindow

```
procedure SetupWindow; virtual;
```

Override: Sometimes

Initializes the scroll bar's range from zero to 100. To override this range, call *TScrollBar.SetRange*.

See also: *TScrollBar.SetRange*

Store

```
procedure Store(var S: TStream);
```

Stores the scroll bar control on the stream *S* by first calling *TControl.Store* and then writing the additional fields (*IsHorizontal*, *LineMagnitude* and *PageMagnitude*) introduced by *TScrollBar*.

See also: *TControl.Store*

Transfer

```
function Transfer(DataPtr: Pointer; TransferFlag: Word): Word; virtual;
```

Override: Sometimes

Transfers the scroll bar's range and current position to or from the memory location pointed to by *DataPtr*. If *TransferFlag* is *tf_GetData*, a special record holding the range and position is transferred to the memory location. If *TransferFlag* is *tf_SetData*, the special record at the memory location retrieved its values are used to set the range and position of the scroll bar. *Transfer* returns the number of bytes stored in or retrieved from the memory location. The special record is defined as follows:

```
ScrollBarTransferRec = record
  LowValue: Integer;
  HighValue: Integer;
  Position: Integer;
end;
```

TScroller WObjects

TScroller objects exist in the *Scroller* field of *TWindow* and descendant objects. The *TScroller* object provides a window an automatic window scrolling mechanism that works in conjunction with horizontal or vertical window scroll bars or without any scroll bars. It supports a technique called auto-scrolling, which does not require scroll bars.

Typically, you will construct and manipulate *TScroller* objects from within the methods of their owner window objects.

Fields

Window `Window: PWindow;` **Read only**

Window points to the *TScroller*'s owner window.

XUnit `XUnit: Integer;` **Read only**

XUnit is the smallest number of device units (pixels) that the window can be horizontally scrolled. *XUnit* values are specified in the *Init* constructor, but can be modified later through method calls.

YUnit `YUnit: Integer;` **Read only**

YUnit is the smallest number of device units (pixels) that the window can be vertically scrolled. *YUnit* values are specified in the *Init* constructor, but can be modified later through method calls.

XPos `XPos: Longint;` **Read only**

XPos is the scroller's current horizontal position in terms of *XUnit* units.

YPos `YPos: Longint;` **Read only**

YPos is the scroller's current vertical position in terms of *YUnit* units.

XRange `XRange: Longint;` **Read only**

XRange is the total number of horizontal *XUnit* units the window can be scrolled. *XRange* values are specified in the *Init* constructor, but can be modifed later.

YRange `YRange: Longint;` **Read only**

YRange is the total number of vertical *YUnit* units the window can be scrolled. *YRange* values are specified in the *Init* constructor, but can be modifed later.

XLine `XLine: Integer;` **Read/write**

XLine is the number of *XUnit* units to scroll horizontally in response to a click on the scroll bar arrow. The default value is 1.

YLine `YLine: Integer;` **Read/write**

YLine is the number of *YUnit* units to scroll vertically in response to a click on the scroll bar arrow. The default value is 1.

XPage `XPage: Integer;` **Read/write**

XPage is the number of *XUnit* units to scroll horizontally in response to a click on the scroll bar's thumb area. By default, *XPage* is equal to the current width of the window in *XUnit* units. Resizing the window updates this value.

YPage `YPage: Integer;` **Read/write**

YPage is the number of *YUnit* units to scroll vertcially in response to a click on the scroll bar's thumb area. By default, *YPage* is equal to the current height of the window in *YUnit* units. Resizing the window updates this value.

AutoMode `AutoMode: Boolean;` **Read/write**

AutoMode is *True* if the scroller is in auto-scrolling mode. By default, *AutoMode* is *True*.

TrackMode `TrackMode: Boolean;` **Read/write**

TrackMode is *True* if the scroller automatically tracks scroll bar thumb movements by scrolling the window. By default, *TrackMode* is *True*.

HasHScrollBar `HasHScrollBar: Boolean;` **Read/write**

If the owner window has a horizontal window scroll bar, *HasHScrollBar* is *True*.

HasVScrollBar `HasVScrollBar: Boolean;` **Read/write**

If the owner window has a vertical window scroll bar, *HasVScrollBar* is *False*.

Methods

Init

```
constructor Init(TheWindow: PWindow; TheXUnit, TheYUnit: Integer;
TheXRange, TheYRange: Longint);
```

Constructs a new *TScroller* object with *TheWindow* as the owner window, and *TheXUnit*, *TheYUnit*, *TheXRange*, and *TheYRange* as *XUnit*, *YUnit*, *XRange* and *YRange*, respectively. Sets *AutoMode* and *TrackMode* to *True* and sets *HasHScrollBar* and *HasVScrollBar* depending on the scroll bar attributes of the owner window.

Load

```
constructor Load(var S: TStream);
```

Constructs and loads a scroller from the stream *S* by first calling *TObject.Init* and then reading the *TScroller* fields, with the exception of *XPage*, *YPage*, *XPos* and *YPos*.

AutoScroll

```
procedure AutoScroll; virtual;
```

Override: Sometimes

Depending on the position of the mouse cursor, performs auto-scrolling.

See also: *TWindow.WMTimer*

BeginView

```
procedure BeginView(PaintDC: HDC; var PaintInfo: TPaintStruct); virtual;
```

Override: Sometimes

Sets the origin of the owner window's paint display context (*PaintDC*) according to the current scroller position.

EndView

```
procedure EndView; virtual;
```

Override: Sometimes

Updates the position of the owner window's scroll bars to be in sync with the position of the *TScroller*.

HScroll

```
procedure HScroll(ScrollRequest: Word; ThumbPos: Integer); virtual;
```

Override: Never

Responds to horizontal scroll bar events by changing the position of the scroller and the horizontal scroll bar.

See also: *TWindow.WMHScroll*

IsVisibleRect

```
function IsVisibleRect(X, Y: Longint; XExt, YExt: Integer): Boolean;
virtual;
```

Override: Seldom

Returns *True* if any portion of the rectangle specified by the passed arguments is currently visible in the owner window.

ScrollBy

```
procedure ScrollBy(Dx, Dy: Longint); virtual;
```

Override: Seldom

Scrolls by the amounts specified by *Dx* and *Dy*. Also updates the window's display.

ScrollTo

```
procedure ScrollTo(X, Y: Longint); virtual;
```

TS

Override: Sometimes Scrolls to the position specified by *X* and *Y*. Also updates the window's display.

SetPageSize `procedure SetPageSize; virtual;`

Override: Sometimes Sets the *XPage* and *YPage* fields to be the owner window's current width and height, respectively.

See also: *TWindow.WMSize*

SetRange `procedure SetRange(TheXRange, TheYRange: Longint); virtual;`

Override: Never Overrides the *XRange* and *YRange* values passed in the call to *Init* by setting them to *TheXRange* and *TheYRange*. Then calls *SetSBarRange* to set the range of the owner window's scroll bars.

See also: *TScroller.SetSBarRange*

SetSBarRange `procedure SetSBarRange; virtual;`

Override: Never Sets the range of the of the owner window's scroll bars to be in sync with the range of the *TScroller*.

See also: *TWindow.SetupWindow*

SetUnits **procedure** SetUnits(TheXUnit, TheYUnit: Longint); **virtual;**

Sets the *XUnit* and *YUnit* fields to *TheXUnit* and *TheYUnit*, respectively.

Store `procedure Store(var S: TStream);`

Stores the scroller on the stream *S* by writing the *TScroller* fields, with the exception of *XPage*, *YPage*, *XPos* and *YPos*.

VScroll `procedure VScroll(ScrollRequest: Word; ThumbPos: Integer); virtual;`

Override: Never Responds to vertical scroll bar events by changing the position of the scroller and the vertical scroll bar.

See also: *TWindow.WMVScroll*

TSortedCollection — WObjects

TSortedCollection is a specialized derivative of *TCollection* implementing collections sorted by key without duplicates. Sorting is implied by a virtual *TSortedCollection.Compare* method which you override to provide your own definition of element ordering. As new items are added they are automatically inserted in the order given by the *Compare* method. Items can be located using the binary search method, *TSortedCollection.Search*.

The virtual *KeyOf* method that returns a pointer for *Compare*, can also be overridden if *Compare* needs additional information.

Field

Duplicates

```
Duplicates: Boolean;
```

The *Duplicates* field determines whether items with duplicate keys will be accepted by the stream. If *Duplicates* is *True*, duplicate-key entries will be inserted into the collection. If *Duplicates* is *False* (the default), a new, duplicate-key item will *replace* the existing item with that key.

Methods

Load

```
constructor Load(var S: TStream);
```

Constructs and loads a sorted collection from the stream *S* by first calling *TCollection.Load* and then reading the *TSortedCollection* field *Duplicates*.

See also: *TCollection.Load*

Compare

```
function Compare(Key1, Key2: Pointer): Integer; virtual;
```

Override: Always

Compare is an abstract method that must be overridden in all descendant types. *Compare* should compare the two key values, and return a result as follows:

–1	if *Key1* < *Key2*
0	if *Key1* = *Key2*
1	if *Key1* > *Key2*

Key1 and *Key2* are pointer values, as extracted from their corresponding collection items by the *TSortedCollection.KeyOf* method. The *TSortedCollection.Search* method implements a binary search through the collection's items using *Compare* to compare the items.

See also: *TSortedCollection.KeyOf*, *TSortedCollection.Compare*

IndexOf

```
function IndexOf(Item: Pointer): Integer; virtual;
```

Override: Never

Uses *TSortedCollection.Search* to find the index of the given *Item*. If the item is not in the collection, *IndexOf* returns –1. The actual implementation of *TSortedCollection.IndexOf* is:

```
if Search(KeyOf(Item), I) then IndexOf := I else IndexOf := -1;
```

See also: *TSortedCollection.Search*

Insert

```
procedure Insert(Item: Pointer); virtual;
```

Override: Never

If the target item is not found in the sorted collection, it is inserted at the correct index position. Calls *TSortedCollection.Search* to determine if the item exists, and if not, where to insert it. The actual implementation of *TSortedCollection.Insert* is:

```
if not Search(KeyOf(Item), I) then AtInsert(I, Item);
```

See also: *TSortedCollection.Search*

KeyOf

```
function KeyOf(Item: Pointer): Pointer; virtual;
```

Override: Sometimes

Given an *Item* from the collection, *KeyOf* should return the corresponding key of the item. The default *TSortedCollection.KeyOf* simply returns *Item*. *KeyOf* is overridden in cases where the key of the item is not the item itself.

See also: *TSortedCollection.IndexOf*

Search

```
function Search(Key: Pointer; var Index: Integer): Boolean; virtual;
```

Override: Seldom

Returns *True* if the item identified by *Key* is found in the sorted collection. If the item is found, *Index* is set to the found index; otherwise *Index* is set to the index where the item would be placed if inserted.

See also: *TSortedCollection.Compare, TSortedCollection.Insert*

Store

```
procedure Store(var S: TStream);
```

Stores the sorted collection and all its items on the stream *S* by calling *TCollection.Store* to write the collection, then writing the *Duplicates* field to the stream.

See also: *TCollection.Store*

TStatic — WObjects

TStatic is an interface object that represents a corresponding static control element in Windows. *TStatic* objects are not normally used in dialog boxes (*TDialog*) or dialog windows (*TDlgWindow*), but are used when you want to display a standalone static control as a child window in another window's client area.

Field

TextLen

```
TextLen: Word;
```

TextLen holds the size of the text buffer for static controls. The number of characters that can actually be stored in the static control is one less than *TextLen*, because of the null terminator on the string. *TextLen* is also the number of bytes transferred by the *Transfer* method.

Methods

Init

```
constructor Init(AParent: PWindowsObject; AnID: Integer; ATitle: PChar;
X, Y, W, H: Integer, ATextLen: Word);
```

Override: Seldom

Constructs a static control object with the passed parent window (*AParent*), control ID (*AnID*), text (*ATitle*), position (*X*, *Y*) relative to the origin of the parent window's client area, width (*W*), height (*H*), and text length(*ATextLen*). The static control will be left justified because *TStatic.Init* adds the Windows style *ss_Left* to the object's *Attr.Style* field. It also removes the *ws_TabStop* style. *Init* then calls *DisableTransfer* to exclude *TStatic* objects from the transfer mechanism, by default.

InitResource

```
constructor InitResource(AParent: PWindowsObject; ResourceID,
 ATextLen: Word);
```

Associates a *TStatic* object with the static control resource specified by *ResourceID*, and sets the *TextLen* field to *ATextLen*. Calls *TControl.InitResource* to construct and associate the object.

See also: *TControl.InitResource*

Load

```
constructor Load(var S: TStream);
```

Constructs and loads a static control from the stream *S*, by calling *TControl.Load*, and then reading the *TextLen* field.

See also: *TControl.Load*

Clear

```
procedure Clear; virtual;
```

Override: Seldom

Clears the static control's text.

GetClassName

```
function GetClassName: PChar; virtual;
```

Override: Seldom

Returns the name of *TStatic*'s window class, 'Static'.

GetText

```
function GetText(ATextString: PChar; MaxChars: Integer): Integer;
virtual;
```

Override: Seldom

Retrieves the static control's text and stores it in the *ATextString* argument. *MaxChars* specifies the maximum size of *ATextString*. *GetText* returns the size of the retrieved string.

SetText

```
procedure SetText(ATextString: PChar); virtual;
```

Override: Seldom

Sets the static control's text to be the string passed in *ATextString*.

Store

```
procedure Store(var S: TStream);
```

Stores the static control on the stream *S* by calling *TControl.Store*, and then writing the *TextLen* field.

See also: *TControl.Store*

Transfer

```
function Transfer(DataPtr: Pointer; TransferFlag: Word): Word; virtual;
```

Override: Sometimes

Transfers *TextLen* characters of the current text of the static control to or from the memory location pointed to by *DataPtr*. If *TransferFlag* is *tf_GetData*, the text is transferred to the memory location. If *TransferFlag* is *tf_SetData*, the static control's text is set to the text at the memory location. Transfer returns *TextLen*, the number of bytes stored in or retrieved from the memory location. If *TransferFlag* is *tf_SizeData*, *Transfer* returns the size of the transfer data.

TStream — WObjects

TStream is a general abstract object providing polymorphic I/O to and from a storage device. You can create your own derived stream objects by overriding the virtual methods: *GetPos*, *GetSize*, *Read*, *Seek*, *Truncate*, and *Write*. ObjectWindows itself does this to derive *TDosStream* and *TEmsStream*. For buffered derived streams, you must also override *TStream.Flush*.

Fields

Status

```
Status: Integer
```

Read/write

Indicates the current status of the stream as shown in Table 5.0.

If *Status* is not *stOk* all operations on the stream are suspended until *Reset* is called.

Table 5.1
Stream error codes

Error code	Meaning
stOk	No error
stError	Access error
stInitError	Cannot initialize stream
stReadError	Read beyond end of stream
stWriteError	Cannot expand stream
stGetError	Get of unregistered object type
stPutError	Put of unregistered object type

ErrorInfo `ErrorInfo: Integer` **Read/write**

ErrorInfo contains additional information when *Status* is not *stOk*. For *Status* values of *stError*, *stInitError*, *stReadError*, and *stWriteError*, *ErrorInfo* contains the DOS or EMS error code, if one is available. When *Status* is *stGetError*, *ErrorInfo* contains the object type ID (the *ObjType* field of a *TStreamRec*) of the unregistered object type. When *Status* is *stPutError*, *ErrorInfo* contains the VMT data segment offset (the *VmtLink* field of a *TStreamRec*) of the unregistered object type.

Methods

CopyFrom `procedure CopyFrom(var S: TStream; Count: Longint);`

Copy *Count* bytes from stream *S* to the calling stream object. For example:

```
{Create a copy of entire stream}
NewStream := New(TEmsStream, Init(OldStream^.GetSize));
OldStream^.Seek(0);
NewStream^.CopyFrom(OldStream, OldStream^.GetSize);
```

See also: *TStream.GetSize*, *TObject.Init*

Error `procedure Error(Code, Info: Integer); virtual;`

Override: Sometimes

Called whenever a stream error occurs. The default *TStream.Error* stores *Code* and *Info* in the *Status* and *ErrorInfo* fields and then, if the global variable *StreamError* is not **nil**, calls the procedure pointed to by *StreamError*. Once an error has occurred, all stream operations on the stream are suspended until *Reset* is called.

See also: *TStream.Reset*, *StreamError* variable

Flush `procedure Flush; virtual;`

Override: Sometimes

An abstract method that must be overridden if your descendant implements a buffer. This method can flush any buffers by clearing the

read buffer, by writing the write buffer, or both. The default *TStream.Flush* does nothing.

See also: *TBufStream.Flush*

Get `function Get: PObject;`

Reads an object from the stream. The object must have been previously written to the stream by *TStream.Put*. *Get* first reads an object type ID (a word) from the stream. It then finds the corresponding object type by comparing the ID to the *ObjType* field of all registered object types (see the *TStreamRec* type), and finally calls the *Load* constructor of that object type to create and load the object. If the object type ID read from the stream is zero, *Get* returns a **nil** pointer; if the object type ID has not been registered (using *RegisterType*), *Get* calls *TStream.Error* and returns a **nil** pointer; otherwise, *Get* returns a pointer to the newly created object.

See also: *TStream.Put, RegisterType, TStreamRec, Load* methods

GetPos `function GetPos: Longint; virtual;`

Override: Always

Returns the value of the calling stream's current position. This is an abstract method that must be overridden.

See also: *TStream.Seek*

GetSize `function GetSize: Longint; virtual;`

Override: Always

Returns the total size of the calling stream. This is an abstract method that must be overridden.

Put `procedure Put(P: PObject);`

Writes an object to the stream. The object can later be read from the stream using *TStream.Get*. *Put* first finds the type registration record of the object by comparing the object's VMT offset to the *VmtLink* field of all registered object types (see the *TStreamRec* type). It then writes the object type ID (the *ObjType* field of the registration record) to the stream, and finally calls the *Store* method of that object type to write the object. If the *P* argument passed to *Put* is **nil**, *Put* writes a word containing zero to the stream. If the object type of *P* has not been registered (using *RegisterType*), *Put* calls *TStream.Error* and doesn't write anything to the stream.

See also: *TStream.Get, RegisterType, TStreamRec, Store* methods

Read `procedure Read(var Buf; Count: Word); virtual;`

Override: Always

This is an abstract method that must be overridden in all descendant types. *Read* should read *Count* bytes from the stream into *Buf* and advance

the current position of the stream by *Count* bytes. If an error occurs, *Read* should call *Error*, and fill *Buf* with *Count* bytes of zero.

See also: *TStream.Write, TStream.Error.*

ReadStr `function ReadStr: PString;`

Reads a string from the current position of the calling stream, returning a *PString* pointer. *TStream.ReadStr* calls *GetMem* to allocate (Length+1) bytes for the string.

See also: *TStream.WriteStr*

Reset `procedure Reset;`

Resets any stream error condition by setting *Status* and *ErrorInfo* to zero. This method lets you continue stream processing following an error condition that you have corrected.

See also: *TStream.Status, TStream.ErrorInfo,* st*XXXX* error codes

Seek `procedure Seek(Pos: Longint); virtual;`

Override: Always

This is an abstract method that must be overridden by all descendants. *TStream.Seek* sets the current position to *Pos* bytes from the start of the calling stream. The start of a stream is position 0.

See also: *TStream.GetPos*

StrRead `function StrRead: PChar;`

Reads a null-terminated string from the stream by first reading the length of the string, and then reading that number of characters. Returns a pointer to the null-terminated string read.

See also: *TStream.StrWrite*

StrWrite `procedure StrWrite(P: PChar);`

Writes the null-terminated string *P* to the stream by first writing the length of the string, and then writing that number of characters.

See also: *TStream.StrRead*

Truncate `procedure Truncate; virtual;`

Override: Always

This is an abstract method that must be overridden by all descendants. *TStream.Truncate* deletes all data on the calling stream from the current position to the end.

See also: *TStream.GetPos, TStream.Seek*

Write `procedure Write(var Buf; Count: Word); virtual;`

Override: Always

This is an abstract method that must be overridden in all descendant types. *Write* should write *Count* bytes from *Buf* onto the stream and advance the current position of the stream by *Count* bytes. If an error occurs, *Write* should call *Error*.

See also: *TStream.Read, TStream.Error.*

WriteStr `procedure WriteStr(P: PString);`

Writes the string *P^* to the calling stream, starting at the current position.

See also: *TStream.ReadStr*

TStrCollection — WObjects

TStrCollection is a simple derivative of *TSortedCollection* implementing a sorted list of ASCII strings. The *TStrCollection.Compare* method is overridden to provide the conventional lexicographic ASCII string ordering. You can override *Compare* to allow for other orderings, such as those for non-English character sets.

Methods

Compare `function Compare(Key1, Key2: Pointer): Integer; virtual;`

Override: Sometimes

Compares the strings *Key1^* and *Key2^* as follows: return –1 if *Key1* < *Key2*; 0 if *Key1* = *Key2*; and +1 if *Key1* > *Key2*.

See also: *TSortedCollection.Search*

FreeItem `procedure FreeItem(Item: Pointer); virtual;`

Override: Seldom

Removes the string *Item^* from the sorted collection and disposes of the string.

GetItem `function GetItem(var S: TStream): Pointer; virtual;`

Override: Seldom

By default, reads a string from the *TStream* by calling *S.ReadStr*.

See also: *TStream.ReadStr*

PutItem `procedure PutItem(var S: TStream; Item: Pointer); virtual;`

Override: Seldom

By default, writes the string *Item^* on to the *TStream* by calling *S.WriteStr*.

See also: *TStream.WriteStr*

TWindow WObjects

TWindow defines the fundamental behavior for all window and control objects. Object instances of *TWindow* are empty generic windows, but they can define menus, cursors and icons.

Fields

Attr

```
Attr: TWindowAttr;
```

Attr holds a *TWindowAttr* record, which defines the windows creation attributes, characteristics that influence the creation of the window object's corresponding interface element. These include the window's associated text, style, extended style, position and size, window handle and control ID. These attributes are traditionally set to defaults in the object's *Init* constructor, but can be overridden in a descendant type's Init constructor. Here is the record definition for *TWindowAttr*:

```
TWindowAttr = record
  Title: PChar;
  Style: Longint;
  ExStyle: Longint;
  X, Y, W, H: Integer;
  Param: Pointer;
  case Integer of
   0: (Menu: HMenu);  { window menu's handle or... }
   1: (Id: Integer);  { control's child identifier }
end;
```

DefaultProc

```
DefaultProc: TFarProc;
```

Read only

Holds the address of the default window procedure, which defines the Windows default processing for Windows messages.

FocusChildHandle

```
FocusChildHandle: THandle;
```

Read only

FocusChildHandle stores the Windows handle to the window's child window that currently had the focus the last time the window was activated.

Scroller

```
Scroller: PScroller;
```

Read/write

Scroller holds a pointer to a *TScroller* object, which is the window's scroller, if any. A scroller should be constructed in the window's *Init* constructor.

Methods

Init

```
constructor Init(AParent: PWindowsObject; ATitle: PChar);
```

Override: Often

Constructs a window object with the parent window passed in *AParent* and the associated text (caption for windows) passed in *ATitle*. *AParent* should be **nil** for main windows, which have no parent. The object's *Attr.Style* field is set to *ws_OverlappedWindow* unless the window is an MDI child window, when the *Attr.Style* field is set to *ws_ClipSiblings*. The position and extent fields in the *Attr* record are set to defaults appropriate for overlapped and popup windows. You can override *TWindow.Init* in your descendant types as long as you first explicitly call *TWindow.Init*. Then you can reset the *Attr* fields. *Scroller*, by default, is set to **nil**.

InitResource

```
constructor InitResource(AParent: PWindowsObject; ResourceID: Word);
```

Constructs an ObjectWindows object to be associated with an interface element (usually a control) created by a resource definition. Calls *TWindowsObject.Init*.

See also: *TWindowsObject.Init*

Load

```
constructor Load(var S: TStream);
```

Constructs and loads a window from the stream *S* by first calling *TWindowsObject.Load* and then reading and getting the additional fields (*Attr* and *Scroller*) introduced by *TWindow*.

See also: *TWindowsObject.Load*

Done

```
destructor Done; virtual;
```

Override: Often

Disposes the *TScroller* object in *Scroller*, if any, before calling *TWindowsObject.Done* to dispose of the entire object.

Create

```
function Create: Boolean; virtual;
```

Creates the window object's corresponding interface element unless the object was constructed with *InitResource*, in which case the interface element already exists. *Create* first calls *Register* to register the window class if not already registered. *Create* then creates the window and calls *SetupWindow*, which you can define to initialize the newly created window, usually by creating child windows and drawing graphics or text. *Create* returns *True* if successful. If unsuccessful, it returns *False* and calls *Error*.

Normally you will never call *Create* directly. *Create* is called by *TApplication.MakeWindow*, which first checks to make sure memory is available.

See also: *TWindowsObject.Register, TWindowsObject.SetupWindow, TWindow.InitResource, TWindowsObject.Error, TApplication.MakeWindow*

DefWndProc

```
procedure DefWndProc(var Msg: TMessage); virtual;
```

Override: Never

Calls the window's default procedure which handles default processing for incoming Windows messages. It stores the result of this call in the *Result* field of the message record, *Msg*.

Destroy

```
procedure Destroy; virtual;
```

Override: Seldom

Simply calls *TWindowsObject.Destroy* to destroy the window element, unless the window is an MDI child window. In that case, *Destroy* sends a *wm_MDIDestroy* message to the window's MDI client window for appropriate default processing.

GetID

```
function GetId: Integer; virtual;
```

Override: Seldom

Returns the window identifier, such as a control ID.

GetWindowClass

```
procedure GetWindowClass(var AWndClass: TWndClass); virtual;
```

Override: Often

Fills a window class record, passed in *AWndClass*, with default values for its registration attributes. The style field is set to *cs_HRedraw* or *cs_VRedraw*. The icon is set to a generic icon and the cursor is set to the stock arrow cursor. The background color is set to the system's window background color. The name of the class to be registered is retrieved through call to *GetClassName*.

See also: *TWindowsObject.GetClassName, TWindowsObject.Register, TWindow.Create*

Paint

```
procedure Paint(PaintDC: HDC; var PaintInfo: TPaintStruct); virtual;
```

Override: Often

Serves as a placeholder for descendant types that define *Paint* methods. *Paint* is called automatically in respond to a request (*wm_Paint*) from Windows to redisplay the window's contents. Use *PaintDC* as the display context. It is already obtained before the call to Paint, and released after *Paint*. The *PaintInfo* paint structure passed contains information about the paint request directly.

See also: *TWindow.WMPaint*

SetupWindow

```
procedure SetupWindow; virtual;
```

Override: Often

Sets up the newly created window. If the window is an MDI child window, *SetupWindow* calls *SetFocus* to give the new window the focus. If the window has a scroller object, *SetupWindow* calls *SetSBarRange* to set the range of its scroll bars.

See also: *TScroller.SetSBarRange*

Store

```
procedure Store(var S: TStream);
```

Stores the window on the stream *S* by first calling *TWindowsObject.Store* and then writing and putting the additional fields (*Attr* and *Scroller*) introduced by *TWindow*.

See also: *TWindowsObject.Store*

WMActivate

```
procedure WMActivate(var Msg: TMessage); virtual wm_First + wm_Activate;
```

Override: Sometimes

For windows that intercept keyboard messages for its controls, responds to the window losing and receiving the focus by saving the handle of the child control that currently has the focus in *FocusChildHandle* and restoring the focus.

See also: *TWindowsObject.EnableKBHandler*

WMCreate

```
procedure WMCreate(var Msg: TMessage); virtual wm_First + wm_Create;
```

Responds to the *wm_Create* message by calling *SetupWindow* and then calling *DefWndProc*. Since window creation is handled differently under ObjectWindows than under Windows, the *wm_Create* message needs to be trapped and used to set up window object attributes.

See also: *TWindow.SetupWindow*

WMDestroy

```
procedure WMDestroy(var Msg: TMessage); virtual wm_First + wm_Destroy;
```

Override: Sometimes

Responds to the window closing by calling *TWindowsObject.WMDestroy* to destroy the window element, unless the window is an MDI child window. In that case, *WMDestroy* sets the window's handle, *HWindow*, to zero and calls the *RemoveChild* method of the window's MDI frame window.

See also: *TWindowsObject.RemoveChild*

WMHScroll

```
procedure WMHScroll(var Msg: TMessage); virtual wm_First + wm_HScroll;
```

Sometimes

For windows with scrollers, responds to horizontal window scroll bar events by calling the scroller's *HScroll* method and *DefWndProc*.

See also: *TScroller.HScroll*

WMLButtonDown

```
procedure WMLButtonDown(var Msg: TMessage); virtual wm_First +
wm_LButtonDown;
```

Override: Sometimes

For windows with auto-scrolling scrollers, responds to a left mouse button click by capturing all future mouse input until the mouse button is released. If you plan to override this method to process mouse clicks, but still plan to use auto-scrolling, be sure to call this method from your *WMLButtonDown* method.

WMPaint

```
procedure WMPaint(var Msg: TMessage); virtual wm_First + wm_Paint;
```

Override: Seldom

Responds to the Windows *wm_Paint* message by calling the window object's *Paint* method. If the window has a scroller, *WMPaint* calls *BeginView* before calling *Paint* and *EndView* after calling *Paint*.

See also: *TWindow.Paint, TScroller.EndView, TScroller.BeginView*

WMSize

```
procedure WMSize(var Msg: TMessage); virtual wm_First + wm_Size;
```

Override: Sometimes

For windows with scrollers, responds to a window sizing event by calling *SetPageSize* to adjust for the new window size.

See also: *TScroller.SetPageSize*

WMVScroll

```
procedure WMVScroll(var Msg: TMessage); virtual wm_First + wm_VScroll;
```

Override: Sometimes

For windows with scrollers, responds to vertical window scroll bar events by calling the scroller's *VScroll* method and *DefWndProc*.

See also: *TScroller.VScroll*

TWindowsObject — WObjects

TWindowsObject defines the fundamental behavior for all interface objects, including windows, dialog boxes and controls. *TWindowsObject* is an abstract object type and its methods are useful only to descendant types. Its methods implement the fundamental interface element creation and destruction behavior, window class registration behavior, and the automatic message response mechanism.

Fields

ChildList

```
ChildList: PWindowsObject;
```

Read only

ChildList is a linked list of all of the interface object's child window objects, such as popup windows, dialog boxes and controls. *ChildList* always points to the object most recently added.

Flags

```
Flags: Byte;
```

Read/write

Flags is a byte of data whose bits are used to store the following windows attributes: keyboard handling, auto-creation, transfer, MDI status, and resource creation. *Flags* contains one or more of the *wb_* constants documented in Chapter 6, "Global reference."

See also: *TWindowsObject.SetFlags, TWindowsObject.IsFlagSet*

HWindow `HWindow: HWnd;` **Read only**

HWindow holds a handle to the interface object's associated interface element. If there is no associated element, *HWindow* is equal to zero, the value of an invalid handle. Upon creation of the associated interface element (*Create*), *HWindow* is set to a new handle. Upon destruction of the associated interface element (*WMNCDestroy*), *HWindow* is set to zero.

Instance `Instance: TFarProc;` **Read only**

Instance points to the code executed prior to entering the shared window procedure of ObjectWindows.

Parent `Parent: PWindowsObject;` **Read only**

PWindowsObject points to the interface object that serves as this interface object's parent window. For example, the *Parent* field of a control object that appears in a window object would point to the window object, its parent.

Status `Status: Integer;` **Read/Write**

Status indicates the success of the current effort to initialize an interface object and its associated interface element. The process is successful so far if *Status* is greater than or equal to zero and unsuccessful if it is negative. Descendants of *TWindowsObject*, including *TWindow* and *TDialog* check *Status* before creating their associated elements. Use *Status* in the code of your descendant types to flag an initialization error.

Possible error values include *em_InvalidWindow*, *em_InvalidClient*, *em_InvalidChild*, and *em_InvalidMainWindow*.

TransferBuffer `TransferBuffer: Pointer;` **Read/write**

TransferBuffer points to a transfer record defined by an application that uses the tranfer mechanism. Otherwise it is **nil**.

Methods

Init

```
constructor Init(AParent: PWindowsObject);
```

Override: Often

Constructs and initializes the interface object. Constructors of descendant types must result in a call to *TWindowsObject.Init*. Calls EnableAutoCreate so that child windows will, by default, be created and displayed along with their parent windows. Also adds the interface object to the child window list of its parent window object.

See also: *TWindowsObject.EnableAutoCreate, TWindowsObject.AddChild*

Load

```
constructor Load(var S: TStream);
```

Constructs and loads an inteface object from the stream *S* by reading the *Status*, other attributes, the size of *ChildList* and then by loading each child window.

Done

```
destructor Done; virtual;
```

Override: Often

Destroys the interface object by first destroying the associated interface element, if any, and calling *TObject.Done*. Disposes all its child windows and removes itself from its parent's child window list. Destructors of descendant types must result in a call to *TWindowsObject.Done*.

CanClose

```
function CanClose: Boolean; virtual;
```

Override: Sometimes

Calls the *CanClose* method for each child window, and returns *False* if any child window returns *False*, indicating that it is not OK to close the interface element. If all child windows' *CanClose* methods return *True*, *CanClose* returns *True*.

ChildWithID

```
function ChildWithId(Id: Integer): PWindowsObject; virtual;
```

Override: Never

Returns a pointer to the window in the child window list with the passed ID. If no child window matches, *ChildWithID* returns **nil**. If *Id* is –1, returns the first non-control object in the child window list.

Create

```
function Create: Boolean; virtual;
```

Override: Never

This is an abstract method which is overridden in descendant types to create the interface object's associated interface element.

DefChildProc

```
procedure DefChildProc(var Msg: TMessage); virtual;
```

Override: Sometimes

Performs the default processing for and incoming child-ID-based message by setting the *Result* field of *Msg* to zero, indicating that the message was not processed.

DefCommandProc

```
procedure DefCommandProc(var Msg: TMessage); virtual;
```

Override: Sometimes

Performs the default processing for and incoming command-based message by setting the *Result* field of *Msg* to zero, indicating that the message was not processed.

DefNotificationProc

```
procedure DefNotificationProc(var Msg: TMessage); virtual;
```

Override: Sometimes

Performs the default processing for and incoming notification message by setting the *Result* field of *Msg* to zero, indicating that the message was not processed.

DefScrollProc

```
procedure DefScrollProc(var Msg: TMessage); virtual;
```

Override: Sometimes

Performs the default processing for and incoming scroll message by setting the *Result* field of *Msg* to zero, indicating that the message was not processed.

DefWndProc

```
procedure DefWndProc(var Msg: TMessage); virtual;
```

Override: Never

This default window procedure does nothing and is usually overridden. It relies on the the *Result* field of *Msg* to remain zero, indicating that the message was not processed. *TWindow* overrides *DefWndProc* to call Windows-supplied default responses to Windows messages.

See also: *TWindow.DefWndProc*

Destroy

```
procedure Destroy; virtual;
```

Override: Never

Forces the destruction of the interface object's associated element, removing it from the screen, by causing the window object to receive a *wm_Destroy* message. Also calls *EnableAutoCreate* for any child window that has been created. This ensures that, if recreated, the interface object will look like it did when destroyed.

See also: *TWindowsObject.WMDestroy*, *TWindowsObject.EnableAutoCreate*

DisableAutoCreate

```
procedure DisableAutoCreate;
```

Disables the feature that allows the interface object, as a child window, to be created and displayed along with its parent window. Call *DisableAutoCreate* for popup windows and controls if you wish to create and display them at a time later than their parent windows.

See also: *TWindowsObject.EnableAutoCreate.*

DisableTransfer

```
procedure DisableTransfer;
```

Disables, for the interface object, the transfer mechanism, which allows a control's state information to be transferred to and from a transfer buffer.

DispatchScroll `procedure DispatchScroll(var Msg: TMessage); virtual;`

Override: Never

Called by *WMHScroll* and *WMVScroll* to dispatch window scrolling messages to the appropriate objects.

See also: *TWindowsObject.WMHScroll, TWindowsObject.WMVScroll*

EnableAutoCreate `procedure EnableAutoCreate;`

Ensures that the interface object, as a child window, is created and displayed along with its parent window. This feature is enabled, by default, for windows and controls, but disabled for dialogs. Call *EnableAutoCreate* if you wish to create a display a dialog along with its parent windows.

See also: *TWindowsObject.DisableAutoCreate*

EnableKBHandler `procedure EnableKBHandler;`

Enables a feature of windows and modeless dialogs that allows them to provide a keyboard interface to child controls, much like that provided by modal dialogs. This allows user to tab through controls, for example. By default, this feature is off.

EnableTransfer `procedure EnableTransfer;`

Enables, for the interface object, the transfer mechanism, which allows a control's state information to be transferred to and from a transfer buffer.

FirstThat `function FirstThat(Test: Pointer): PWindowsObject;`

Iterates over the child window list and calls the Boolean function pointed to by *Test*, passing each child window object in turn as an argument. *FirstThat* returns a pointer to the first child window object that returns *True* from the Boolean function (or **nil** if none returns *True*) and then stops iterating. For example, you can write a method, *GetFirstChecked*, that uses *FirstThat* to retrieve the first child check box in a checked state:

```
function MyWindow.GetFirstChecked: PWindowsObject;

  function IsThisOneChecked(ABox: PWindowsObject); Boolean; far;
  begin
    IsThisOneChecked := ABox^.GetCheck <> 0;
  end;

begin
  GetFirstChecked := FirstThat(@IsThisOneChecked);
end;
```

ForEach `procedure ForEach(Action: Pointer);`

Iterates over the child window list and, for each child window, calls the procedure pointed to by *Action* and passes the child window object as an argument. For example, you can write a method, *CheckAllBoxes*, that uses *ForEach* to check every check box in the child window list:

```
procedure MyWindow.CheckAllBoxes;

  procedure CheckTheBox(ABox: PWindowsObject); far;
  begin
    PCheckBox(ABox)^.Check;
  end;

begin
  ForEach(@CheckAllBoxes);
end;
```

GetChildPtr

```
procedure GetChildPtr(var S: TStream; var P);
```

Loads a child window pointer *P* from the stream *S*. *GetChildPtr* should only be used inside a *Load* constructor to read pointer values that where written by a call to *PutChildPtr* from a *Store* method.

See also: *TWindowsObject.GetSiblingPtr, TWindowsObject.PutSiblingPtr, TWindowsObject.PutChildPtr*

GetClassName

```
function GetClassName: PChar; virtual;
```

Override: Sometimes

Returns the default window class name, 'TurboWindow'.

GetClient

```
function GetClient: PMDIClient; virtual;
```

Override: Never

Returns **nil** for all non-MDI interface objects, which have no MDI client windows. *TMDIWindow* overrides this method to supply its MDI client window.

GetID

```
function GetId: Integer; virtual;
```

Override: Seldom

In general, *GetID* is used to return the window identifier. By default, *TWindowsObject.GetID* simply returns –1. *GetId* is redefined by *TControl* to return the object's control ID. All other interface objects have no control IDs.

See also: *TControl.GetId*

GetSiblingPtr

```
procedure GetSiblingPtr(var S: TStream; var P);
```

Loads a sibling window pointer *P* from the stream *S*. A *sibling window* is a window with the same parent as this window—a *TCheckBox*'s *TGroupBox*, for example, is a sibling of the *TCheckBox* in a dialog. *GetSiblingPtr* should only be used inside a *Load* constructor to read pointer values that where written by a call to *PutSiblingPtr* from a *Store* method. The value loaded

into *P* does not become valid until the window's parent completes it's *Load* operation; therefore, de-referencing a sibling window pointer within a *Load* constructor does not produce the correct result.

See also: *TWindowsObject.PutSiblingPtr, TWindowsObject.GetChildPtr, TWindowsObject.PutChildPtr*

GetWindowClass

```
procedure GetWindowClass(var AWndClass:TWndClass); virtual;
```

Override: Sometimes

Serves as a place holder for descendant types to define the window class record and return it in *AWndClass*. This *GetWindowClass* does nothing.

IsFlagSet

```
function IsFlagSet(Mask: Byte); Boolean;
```

Returns the state of the bit flag in the *Flags* field specified by value in *Mask*. *IsFlagSet* returns *True* if the bit flag is sey on, and *False* if it is off.

See also: *TWindowsObject.SetFlags*

Next

```
function Next: PWindowsObject;
```

Returns a pointer to the next object in the parent window's child window list.

See also: *TWindowsObject.Previous*

Previous

```
function Previous: PWindowsObject;
```

Returns a pointer to the previous object in the parent window's child window list.

See also: *TWindowsObject.Next*

PutChildPtr

```
procedure PutSibingPtr(var S: TStream; var P: PWindowsObject);
```

Store a child window pointer *P* on the stream *S*. *PutChildPtr* should only be used inside a *Store* method to write pointer values that can later be read by a call to *GetChildPtr* from a *Load* constructor.

See also: *TWindowsObject.GetSiblingPtr, TWindowsObject.PutSiblingPtr, TWindowsObject.GetChildPtr*

PutSiblingPtr

```
procedure PutSiblingPtr(var S: TStream; P: PWindowsObject);
```

Stores a sibling pointer *P* on the stream *S*. A sibling window is a window with the same parent as this window. *PutSiblingWindow* should only be used inside a *Store* method to write pointer values that can later be read by a call to *GetSiblingPtr* from a *Load* constructor.

See also: *TWindowsObject.GetSiblingPtr, TWindowsObject.GetChildPtr, TWindowsObject.PutChildPtr*

Register

```
function Register: Boolean; virtual;
```

Override: Never

Registers the window class defined in the object's *GetWindowClass* method and named in its *GetClassName* method if it is not already registered. *Register* returns *True* if the class is registered successfully.

SetFlags

```
procedure SetFlags(Mask: Byte; OnOff: Boolean);
```

Turns a bit flag in the *Flags* field on or off, depending on the value of *OnOff*. If *OnOff* is *True*, the bit in *Mask* is set. Otherwise, the bit is cleared. *Mask* can be any of the *wb_* constants, or a combination of them.

See also: *TWindowsObject.IsFlagSet*

SetupWindow

```
procedure SetupWindow; virtual;
```

Override: Often

Performs initialization of the newly-created interface element, usually by creating child window elements, if any. It creates only those child windows whose auto-create feature is enabled. By default, this includes windows and controls, but not dialogs. If *Create* is unable to create a child window, it sets *Status* to *em_InvalidChild*. *SetupWindow* also calls *TransferData* to copy data into the new child windows.

Show

```
procedure Show(ShowCmd: Integer); virtual
```

Override: Never

Show displays the interface element on the screen in a manner specified by the value passed in *ShowCmd*. The allowable values for *ShowCmd* include:

Value	Description
sw_Hide	Hidden.
sw_Show	In the window's current size and position.
sw_ShowMaximized	Maximized and active.
sw_ShowMinimized	Minimized and active.
sw_ShowNormal	Restored and active.

Store

```
procedure Store(var S: TStream);
```

Stores the interface object on the stream *S* by writing the *Status*, other attributes, and the size of *ChildList*. Each child window is then stored.

Transfer

```
function Transfer(DataPtr: Pointer; TransferFlag: Word): Word; virtual;
```

Override: Sometimes

Returns zero. Redefined by *TControl* descendants to transfer their state data to and from transfer buffer. The return value from *Transfer* is the number of bytes of data transferred.

TransferData

```
procedure TransferData(Direction: Word); virtual;
```

Override: Sometimes

If the transfer mechanism is enabled, by setting *TransferBuffer* to a transfer record, transfers data from to or from the buffer and the interface object's participating child windows. *TransferData* calls the *Transfer* method of

each participating child window and passes a pointer to the transfer buffer as well as the direction specified in *Direction*. *Direction* can be *tf_SetData* or *tf_GetData*.

See also: *TWindowsObject.EnableTransfer*, *TWindowsObject.DisableTransfer*, *TWindowsObject.SetupWindow*

WMActivate

```
procedure WMActivate(var Msg: TMessage); virtual wm_First + wm_Activate;
```

Override: Sometimes

In the case that the interface object is participating in the keyboard handling mechanism, responds to the interface object becoming the active window by calling the application object's *SetKeyboardHandler* method.

See also: *TApplication.SetKeyboardHandler*

WMClose

```
procedure WMClose(var Msg: TMessage); virtual wm_First + wm_Close;
```

Override: Sometimes

Responds to a request to close the window by calling this object's *CanClose* method, or the application object's *CanClose* method in the case that this object is the application's main window. If *CanClose* returns *True*, this interface element is destroyed by calling *Destroy*.

See also: *TWindowsObject.Destroy*

WMCommand

```
procedure WMCommand(var Msg: TMessage); virtual wm_First + wm_Command;
```

Override: Seldom

Provides the mechanism for handling command-based and child-ID-based messages and calling the appropriate response methods.

WMDestroy

```
procedure WMDestroy(var Msg: TMessage); virtual wm_First + wm_Destroy;
```

Override: Seldom

In the case that this object is the application's main window, *WMDestroy* responds to this interface element's destruction by informing Windows that the application is ending. A *wm_Quit* message results. If the object is not the application's main window, the default window behavior is invoked.

WMHScroll

```
procedure WMHScroll(var Msg: TMessage); virtual wm_First + wm_HScroll;
```

Override: Seldom

Intercepts horizontal window scroll bar messages and calls *DispatchScroll*.

See also: *TWindowsObject.DispatchScroll*

WMNCDestroy

```
procedure WMNCDestroy(var Msg: TMessage); virtual wm_First +
wm_NCDestroy;
```

Override: Never

Responds to the last message an interface element receives before destruction by setting *HWindow* to zero.

WMVScroll `procedure WMVScroll(var Msg: TMessage); virtual wm_First + wm_VScroll;`

Override: Seldom

Intercepts vertical window scroll bar messages and calls *DispatchScroll*.

See also: *TWindowsObject.DispatchScroll*

C H A P T E R

6

Global reference

This chapter describes all the elements of ObjectWindows that are *not* part of the ObjectWindows standard object hierarchy. The standard objects are all described in Chapter 5, "Object Windows reference."

The elements listed in this chapter include types, constants, variables, procedures, and functions defined in the ObjectWindows units. A typical entry looks like this:

Sample procedure — Sample's unit

Declaration `procedure Sample(AParameter);`

Function *Sample* performs some useful function on its parameter, *AParameter*.

See also *Example* function

Abstract procedure — WObjects

Declaration `procedure Abstract;`

Function A call to this procedure terminates the program with a run-time error 211. When implementing an *abstract object type*, use calls to *Abstract* in those virtual methods that must be overridden in descendant types. This ensures that any attempt to use instantances of the abstract object type will fail.

See also "Abstract methods" in Chapter 7 of the *Windows Programming Guide*

AllocMultiSel function — WObjects

Declaration `function AllocMultiSel(Count: Integer): PMultiSelRec;`

Function Allocates a *TMultiSelRec* with the count equal to *Count*, and enough room in the *Selections* field to hold *Count* selections (0..*Count*-1). Returns **nil** if there is not enough memory to allocate the entire record.

See also *FreeMultiSel, TMultiSelRec*

Application variable — WObjects

Declaration `Application: PApplication = nil;`

Function The *Application* variable is set to *@Self* at the beginning of *TApplication.Init* and cleared to **nil** by *TApplication.Done*. Thus, throughout the execution of an ObjectWindows program, *Application* points to the application object.

See also *TApplication.Init*

bf_*XXXX* constants — WObjects

Function Button, check box, and radio button objects use *bf_* constants to define their three possible states.

Values The following button flag constants are defined:

Table 6.1 Button flag constants

Constant	Value	Meaning
bf_Unchecked	0	Item is unchecked.
bf_Checked	1	Item is checked.
bf_Grayed	2	Item is grayed.

cm_XXXX constants — WObjects

Function ObjectWindows defines several constants defining ranges of command message constants.

Values The following command constants are defined:

Table 6.2 Command message constants

Constant	Value	Meaning
cm_First	$A000	Beginning of command messages
cm_Count	$6000	Number of command messages
cm_Internal	$FF00	Beginning of command messages reserved for internal use
cm_InternalOffset		*cm_Internal – cm_First*

*cm_* constants are defined for three standard menus: File, Edit, and Window:

Table 6.3 Command offset based default values

Constant	Value	Menu equivalent
cm_EditCut	*cm_InternalOffset*	Edit | Cut
cm_EditCopy	*cm_InternalOffset* + 1	Edit | Copy
cm_EditPaste	*cm_InternalOffset* + 2	Edit | Paste
cm_EditDelete	*cm_InternalOffset* + 3	Edit | Delete
cm_EditClear	*cm_InternalOffset* + 4	Edit | Clear
cm_EditUndo	*cm_InternalOffset* + 5	Edit | Undo
cm_FileNew	*cm_InternalOffset* + 6	File | New
cm_FileOpen	*cm_InternalOffset* + 7	File | Open
cm_MDIFileNew	*cm_InternalOffset* + 8	File | New
cm_MDIFileOpen	*cm_InternalOffset* + 9	File | Open
cm_FileSave	*cm_InternalOffset* + 10	File | Save
cm_FileSaveAs	*cm_InternalOffset* + 11	File | Save As
cm_ArrangeIcons	*cm_InternalOffset* + 12	Window | Arrange Icons
cm_TileChildren	*cm_InternalOffset* + 13	Window | Tile
cm_CascadeChildren	*cm_InternalOffset* + 14	Window | Cascade
cm_CloseChildren	*cm_InternalOffset* + 15	Window | Close All

coXXXX constants — WObjects

Function The *coXXXX* constants are passed as the *Code* parameter to the *TCollection.Error* method when a *TCollection* detects an error during an operation.

Values The following standard error codes are defined for all ObjectWindows collections:

Table 6.4 Collection error codes

Error code	Value	Meaning
coIndexError	–1	Index out of range. The *Info* parameter passed to the *Error* method contains the invalid index.
coOverflow	–2	Collection overflow. *TCollection.SetLimit* failed to expand the collection to the requested size. The *Info* parameter passed to the *Error* method contains the requested size.

See also *TCollection* object

em_XXXX constants — WObjects

Function Several standard error conditions are flagged by ObjectWindows constants starting with *em_*.

Values The following error flags are defined:

Table 6.5 Error condition constants

Constant	Value	Meaning
em_OutOfMemory	–1	A memory allocation ate into the safety pool.
em_InvalidClient	–2	MDI client window could not be created.
em_InvalidChild	–3	One or more of the window's children is not valid.
em_InvalidWindow	–4	Window is invalid because *Create* did not succeed.
em_InvalidMainWindow	–5	Main window could not be created.

EmsCurHandle variable — WObjects

Declaration `EmsCurHandle: Word = $FFFF;`

Function Holds the current EMS handle as mapped into EMS physical page 0 by a *TEmsStream*. *TEmsStream* avoids costly EMS remapping calls by caching the state of EMS. If your program uses EMS for other purposes, be sure to set *EmsCurHandle* and *EmsCurPage* to $FFFF before using a *TEmsStream*—this will force the *TEmsStream* to restore its mapping.

See also *TEmsStream.Handle*

EmsCurPage variable — WObjects

Declaration `EmsCurPage: Word = $FFFF;`

Function Holds the current EMS logical page number as mapped into EMS physical page 0 by a *TEmsStream*. *TEmsStream* avoids costly EMS remapping calls by caching the state of EMS. If your program uses EMS for other purposes, be sure to set *EmsCurHandle* and *EmsCurPage* to $FFFF before using a *TEmsStream*—this will force the *TEmsStream* to restore its mapping.

See also *TEmsStream.Page*

FreeMultiSel procedure — WObjects

Declaration `procedure FreeMultiSel(P: PMultiSelRec);`

Frees a *TMultiSelRec* record allocated by *AllocMultiSel*.

See also *AllocMultiSel*, *TMultiSelRec*

id_XXXX constants — WObjects

Function ObjectWindows defines several constants defining ranges of child ID messages.

Values The following child ID message constants are defined:

Table 6.6 Child ID message constants

Constant	Value	Meaning
id_First	$8000	Start of child ID messages
id_Count	$1000	Number of child ID messages
id_Internal	$8F00	Reserved for internal use
id_InternalOffset	*id_Internal – id_First;*	

Constant	Value	Meaning
id_FirstMDIChild	*id_InternalOffset* + 1	Base for child-ID numbers
id_MDIClient	*id_InternalOffset* + 2	Child-ID number of the MDI client window

LongDiv function — WObjects

Declaration
```
function LongDiv(X: Longint; Y: Integer): Integer;
inline($59/$58/$5A/$F7/$F9);
```

Function A fast, inline assembly-coded division routine, returning the integer value `X/Y`.

LongMul function — WObjects

Declaration
```
function LongMul(X, Y: Integer): Longint;
inline($5A/$58/$F7/$EA);
```

Function A fast, inline assembly-coded multiplication routine, returning the long integer value `X * Y`.

LongRec type — WObjects

Declaration

```
LongRec = record
  Lo, Hi: Word;
end;
```

Function A useful record type for handling double-word length variables.

LowMemory function — WObjects

Declaration

```
function LowMemory: Boolean;
```

Function The *LowMemory* function returns *True* if a memory allocation has eaten into the safety pool at the end of the heap. The size of the safety pool is determined by the *SafetyPoolSize* variable. *LowMemory* is checked automatically by *TApplication.MakeWindow* and *TApplication.ExecDialog*, which should be used for creating window elements. Major consumers of memory (such as large, complex dialog boxes) should check *LowMemory* for themselves periodically to ensure that they don't exceed the available space.

For more details on the use of the safety pool, see "Writing safe programs" in Chapter 19 of the *Windows Programming Guide*.

See also *AllocMem, SafetyPoolSize, TApplication.ValidWindow*

MaxCollectionSize variable — WObjects

Declaration

```
MaxCollectionSize = 65520 div SizeOf(Pointer);
```

Function *MaxCollectionSize* determines that maximum number of elements that may be contained in a collection, which is essentially the number of pointers that can fit in a 64K memory segment.

nf_XXXX constants — WObjects

Function ObjectWindows defines several constants establishing ranges of notification messages.

Values The following constants are defined:

Table 6.7 Notification message constants

Constant	Value	Meaning
nf_First	$9000	Beginning of notification messages
nf_Count	$1000	Number of notification messages
nf_Internal	$9F00	Beginning of notification messages reserved for internal use

PString type — WObjects

Declaration
```
PString = ^String;
```

Function Defines a pointer to a Pascal string.

PtrRec type — WObjects

Declaration
```
PtrRec = record
  Ofs, Seg: Word;
end;
```

Function A record holding the offset and segment values of a pointer.

RegisterType procedure — WObjects

Declaration
```
procedure RegisterType(var S: TStreamRec);
```

Function A ObjectWindows object type must be registered using this method before it can be used in stream I/O. The standard object types are preregistered with *ObjType* values in the reserved range 0..99. *RegisterType* creates an entry in a linked list of *TStreamRec* records.

See also *TStream.Get, TStream.Put, TStreamRec*

SafetyPoolSize variable — WObjects

Declaration `SafetyPoolSize: Word = 8192;`

Defines the size of the memory safety pool. The safety pool is a buffer at the high end of the heap used to ensure that memory allocations do not fail. Use of the safety pool is described in Chapter 19 of the *Windows Programming Guide*.

See also *AllocMem, LowMemory, TApplication.ValidWindow*

StrDispose procedure — WObjects

Declaration `procedure StrDispose(P: PChar);`

Disposes of a string allocated on the heap by the *StrNew* function.

See also *StrNew*

StrNew function — WObjects

Declaration `function StrNew(S: PChar): PChar;`

Function Dynamic string routine. If *S* is nul, *StrNew* returns a **nil** pointer; otherwise, *Length(S)*+1 bytes is allocated containing a copy of *S*, and a pointer to the first byte is returned.

Strings created with *StrNew* should be disposed of with *StrDispose*.

See also *StrDispose*

st*XXXX* constants — WObjects

Function There are two sets of constants beginning with "st" that are used by the ObjectWindows streams system.

Values The following mode constants are used by *TDosStream* and *TBufStream* to determine the file access mode of a file being opened for an ObjectWindows stream:

Table 6.8 Stream access modes

Constant	Value	Meaning
stCreate	$3C00	Create new file
stOpenRead	$3D00	Open existing file with read access only
stOpenWrite	$3D01	Open existing file with write access only
stOpen	$3D02	Open existing file with read and write access

The following values are returned by *TStream.Error* in the *TStream.ErrorInfo* field when a stream error occurs:

Table 6.9 Stream error codes

Error code	Value	Meaning
stOk	0	No error
stError	–1	Access error
stInitError	–2	Cannot initialize stream
stReadError	–3	Read beyond end of stream
stWriteError	–4	Cannot expand stream
stGetError	–5	Get of unregistered object type
stPutError	–6	Put of unregistered object type

See also *TStream*

StreamError variable — WObjects

Declaration

```
StreamError: Pointer = nil;
```

Function If non-**nil**, *StreamError* points to a procedure that will be called by a stream's *Error* method when a stream error occurs. The procedure must be a **far** procedure with one **var** parameter that is a *TStream*. That is, the procedure must be declared as

```
procedure MyStreamErrorProc(var S: TStream); far;
```

StreamError allows you to globally override all stream error handling. To change error handling for a particular type of stream you should override that stream type's *Error* method.

TByteArray type — WObjects

Declaration
```
TByteArray = array[0..32767] of Byte;
```

Function A byte array type for general use in typecasts.

TDialogAttr type — WObjects

Declaration
```
TDialogAttr = record
  Name: PChar;
  Param: Longint;
end;
```

Function *TDialog* objects store their attribute values in a record of type *TDialogAttr*.

See also *TDialog.Attr*

tf_*XXXX* constants — WObjects

Function The *Transfer* method uses flag constants beginning with *tf_*.

Values The following constants are defined:

Table 6.10 Transfer function constants

Constants	Value	Meaning
tf_SizeData	0	Find out the size of data transferred by the object.
tf_GetData	1	Retrieve data from the object.
tf_SetData	2	Send data to set the value of the object.

TItemList type — WObjects

Declaration
```
TItemList = array[0..MaxCollectionSize - 1] of Pointer;
```

Function An array of generic pointers used internally by *TCollection* objects.

TMessage type — WObjects

Declaration

```
TMessage = record
  Receiver: HWnd;
  Message: Word;
  case Integer of
    0: (WParam: Word;
        LParam: Longint;
        Result: Longint);
    1: (WParamLo: Byte;
        WParamHi: Byte;
        LParamLo: Word;
        LParamHi: Word;
        ResultLo: Word;
        ResultHi: Word);
  end;
```

Function The message processing loop in *TApplication* packages Windows message information into *TMessage* records before passing the information along to the appropriate message response method.

See also *TApplication.MessageLoop*

TMultiSelRec type — WObjects

Declaration

```
TMultiSelRec = record
  Count: Integer;
  Selections: array[0..0] of Integer;
end;
```

TMultiSelRec holds a list of selected items for transfer to or from a multiple-selection list box. *Count* indicates the number of selected items, and *Selections* is an open-ended array of integers. Using *AllocMultiSel*, you can allocate a record with enough selection items to accommodate as many selected items as the list box has.

See also *AllocMultiSel, FreeMultiSel*

TStreamRec type WObjects

Declaration

```
TStreamRec = record
  ObjType: Word;
  VmtLink: Word;
  Load: Pointer;
  Store: Pointer;
  Next: Word;
end;
```

Function

An ObjectWindows object type must have a registered *TStreamRec* before its objects can be loaded or stored on a *TStream* object. The *RegisterTypes* routine registers an object type by setting up a *TStreamRec* record.

The fields in the stream registration record are defined as follows:

Table 6.11
Stream record fields

Field	Contents
ObjType	A unique numerical id for the object type
VmtLink	A link to the object type's virtual method table entry
Load	A pointer to the object type's *Load* constructor
Store	A pointer to the object type's *Store* method
Next	A pointer to the next *TStreamRec*

ObjectWindows reserves object type IDs (*ObjType*) values 0 through 999 for its own use. Programmers can define their own values in the range 1,000 to 65,535.

By convention, a *TStreamRec* for a *Txxxx* object type is called *Rxxxx*. For example, the *TStreamRec* for a *TCalculator* type is called *RCalculator*, as shown in the following code:

```
type
  TCalculator = object(TDialog)
    constructor Load(var S: TStream);
    procedure Store(var S: TStream);
    ...
  end;

const
  RCalculator: TStreamRec = (
    ObjType: 2099;
    VmtLink: Ofs(TypeOf(TCalculator)^);
    Load: @TCalculator.Load;
    Store: @TCalculator.Store);
```

```
begin
  RegisterType(RCalculator);
  ...
end;
```

See also *RegisterType*

TWindowAttr type — WObjects

Declaration

```
TWindowAttr = record
  Title: PChar;
  Style: Longint;
  ExStyle: Longint;
  X, Y, W, H: Integer;
  Param: Pointer;
  case Integer of
    0: (Menu: HMenu);                    { window menu's handle or... }
    1: (Id: Integer);                    { control's child identifier }
  end;
```

Function *TWindow* objects define their attributes in *TWindowAttr* records.

See also *TWindow.Attr*

TWordArray type — WObjects

Declaration

```
TWordArray = array[0..16383] of Word;
```

Function A word array type for general use.

wb_*XXXX* constants — WObjects

Function The *Flags* field in *TWindowsObject* is a bitmapped field. The bits can be accessed with constants beginning with *wb_*.

Values The following values are defined:

Table 6.12 *TWindowsObject* bitmapped field constants

Constant	Value	Meaning if set
wb_KeyboardHandler	$01	Window handles key events like a dialog
wb_FromResource	$02	Dialog was loaded from a resource
wb_AutoCreate	$04	Window is created when parent window is created
wb_MDIChild	$08	Window is an MDI child window
wb_Transfer	$10	Window participates in the *Transfer* mechanism. By default, this bit is set by *InitResource*, cleared by *Init*.

See also *TWindowsObject.Flags*

wm_XXXX constants — WObjects

Function ObjectWindows defines several constants related to the standard Windows messages, defining ranges of messages reserved for Windows.

Values The following constants are defined:

Table 6.13 Window message constants

Constant	Value	Meaning
wm_First	$0000	Beginning of Windows messages
wm_Count	$8000	Number of Windows messages

See also Chapter 3, "Windows message reference"

WordRec type — WObjects

Declaration

```
WordRec = record
  Lo, Hi: Byte;
end;
```

Function A utility record allowing access to the *Lo* and *Hi* bytes of a word.

See also *LongRec*

I N D E X

A

B

C

D

E

F

G

H

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y