3 User32.lib

3.1 ActivateKeyboardLayout

The ActivateKeyboardLayout function sets the input locale identifier (formerly called the keyboard layout handle) for the calling thread or the current process. The input locale identifier specifies a locale as well as the physical layout of the keyboard.

```
ActivateKeyboardLayout: procedure
(
    hkl             :dword;
    Flags           :dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__ActivateKeyboardLayout@8" );
```

Parameters

hkl

[in] Input locale identifier to be activated.

Windows 95/98: This parameter can be obtained using LoadKeyboardLayout or GetKeyboardLayoutList, or it can be one of the values in the table that follows.

Windows NT: The input locale identifier must have been loaded by a previous call to the LoadKeyboardLayout function. This parameter must be either the handle to a keyboard layout or one of the following values.

Value	Meaning
HKL_NEXT	Selects the next locale identifier in the circular list of loaded locale identifiers maintained by the system.
HKL_PREV	Selects the previous locale identifier in the circular list of loaded locale identifiers maintained by the system.

Flags

[in] Specifies how the input locale identifier is to be activated. This parameter can be one of the following values.

Value	Meaning

KLF_REORDER	If this bit is set, the system's circular list of loaded locale identifiers is reordered by moving the locale identifier to the head of the list. If this bit is not set, the list is rotated without a change of order. For example, if a user had an English locale identifier active, as well as having French, German, and Spanish locale identifiers loaded (in that order), then activating the German locale identifier with the KLF_REORDER bit set would produce the following order: German, English, French, Spanish. Activating the German locale identifier without the KLF_REORDER bit set would produce the following order: German, Spanish, English, French. If less than three locale identifiers are loaded, the value of this flag is irrelevant.
KLF_RESET	Windows 2000: If set but KLF_SHIFTLOCK is not set, the Caps Lock state is turned off by pressing the Caps Lock key again. If set and KLF_SHIFTLOCK is also set, the Caps Lock state is turned off by pressing either SHIFT key. These two methods are mutually exclusive, and the setting persists as part of the User's profile in the registry.
KLF_SETFORPROCESS	Windows 2000: Activates the specified locale identifier for the entire process and sends the WM_INPUTLANGCHANGE message to the current thread's Focus or Active window.
KLF_SHIFTLOCK	Windows 2000: This is used with KLF_RESET. See KLF_RESET for an explanation.
KLF_UNLOADPREVIOUS	This flag is unsupported. Use the UnloadKeyboardLayout function instead.

Return Values

Windows NT 3.51 and earlier: The return value is of type BOOL. If the function succeeds, it is nonzero. If the function fails, it is zero.

Windows 95/98, Windows NT 4.0 and later: The return value is of type HKL. If the function succeeds, the return value is the previous input locale identifier. Otherwise, it is zero.

To get extended error information, use the GetLastError function.

Remarks

This function is not restricted to keyboard layouts. The hkl parameter is actually an input locale identifier. This is a broader concept than a keyboard layout, since it can also encompass a speech-to-text converter, an IME, or any other form of input. Several input locale identifiers can be loaded at any one time, but only one is active at a time. Loading multiple input locale identifiers makes it possible to rapidly switch between them.

Windows 95/98: An application can create a valid input locale identifier by setting the high word to zero and the low word to a locale identifier. Using such an input locale identifier changes the input language without affecting the physical layout.

When multiple input method editors (IMEs) are allowed for each locale, passing an input locale identifier in which the high word (the device handle) is zero activates the first IME in the list belonging to the locale.

Windows 2000: The KLF_RESET and KLF_SHIFTLOCK flags alter the method by which the Caps Lock state is turned off. By default, the Caps Lock state is turned off by hitting the Caps Lock key again. If only KLF_RESET is set, the default state is reestablished. If KLF_RESET and KLF_SHIFTLOCK are set, the Caps Lock state is turned off by pressing either Caps Lock key. This feature is used to conform to local keyboard behavior standards as well as for personal preferences.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

See Also

Keyboard Input Overview, Keyboard Input Functions, LoadKeyboardLayout, GetKeyboardLayoutName, UnloadKeyboardLayout

---

3.2 AdjustWindowRect

The AdjustWindowRect function calculates the required size of the window rectangle, based on the desired client-rectangle size. The window rectangle can then be passed to the CreateWindow function to create a window whose client area is the desired size.

To specify an extended window style, use the AdjustWindowRectEx function.

```
AdjustWindowRect: procedure
(
    var lpRect      :RECT;
        dwStyle     :dword;
        bMenu       :boolean
);
    stdcall;
    returns( "eax" );
    external( "__imp__AdjustWindowRect@12" );
```

Parameters

lpRect

[in/out] Pointer to a RECT structure that contains the coordinates of the top-left and bottom-right corners of the desired client area. When the function returns, the structure contains the coordinates of the top-left and bottom-right corners of the window to accommodate the desired client area.

dwStyle

[in] Specifies the window style of the window whose required size is to be calculated. Note that you cannot specify the WS_OVERLAPPED style.

bMenu

[in] Specifies whether the window has a menu.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

A client rectangle is the smallest rectangle that completely encloses a client area. A window rectangle is the smallest rectangle that completely encloses the window, which includes the client area and the nonclient area.

The AdjustWindowRect function does not add extra space when a menu bar wraps to two or more rows.

The AdjustWindowRect function does not take the WS_VSCROLL or WS_HSCROLL styles into account. To account for the scroll bars, call the GetSystemMetrics function with SM_CXVSCROLL or SM_CYHSCROLL.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.3 AdjustWindowRectEx

The AdjustWindowRectEx function calculates the required size of the window rectangle, based on the desired size of the client rectangle. The window rectangle can then be passed to the CreateWindowEx function to create a window whose client area is the desired size.

```
AdjustWindowRectEx: procedure
(
    var lpRect      :RECT;
        dwStyle     :dword;
        bMenu       :boolean;
        dwExStyle   :dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__AdjustWindowRectEx@16" );
```

Parameters

lpRect

[in/out] Pointer to a RECT structure that contains the coordinates of the top-left and bottom-right corners of the desired client area. When the function returns, the structure contains the coordinates of the top-left and bottom-right corners of the window to accommodate the desired client area.

dwStyle

[in] Specifies the window style of the window whose required size is to be calculated. Note that you cannot specify the WS_OVERLAPPED style.

bMenu

[in] Specifies whether the window has a menu.

dwExStyle

[in] Specifies the extended window style of the window whose required size is to be calculated. For more information, see CreateWindowEx.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

A client rectangle is the smallest rectangle that completely encloses a client area. A window rectangle is the smallest rectangle that completely encloses the window, which includes the client area and the nonclient area.

The AdjustWindowRectEx function does not add extra space when a menu bar wraps to two or more rows.

The AdjustWindowRectEx function does not take the WS_VSCROLL or WS_HSCROLL styles into account. To account for the scroll bars, call the GetSystemMetrics function with SM_CXVSCROLL or SM_CYHSCROLL.

3.4 AnimateWindow

The AnimateWindow function enables you to produce special effects when showing or hiding windows. There are three types of animation: roll, slide, and alpha-blended fade.

```
AnimateWindow: procedure
(
    hwnd            :dword;
    dwTime          :dword;
    dwFlags         :dword
);
    stdcall;
    returns( "eax" );
    external( "__imp__AnimateWindow@12" );
```

Parameters

hwnd

[in] Handle to the window to animate. The calling thread must own this window.

dwTime

[in] Specifies how long it takes to play the animation, in milliseconds. Typically, an animation takes 200 milliseconds to play.

dwFlags

[in] Specifies the type of animation. This parameter can be one or more of the following values.

Value	Description
AW_SLIDE	Uses slide animation. By default, roll animation is used. This flag is ignored when used with AW_CENTER.
AW_ACTIVATE	Activates the window. Do not use this value with AW_HIDE.
AW_BLEND	Uses a fade effect. This flag can be used only if hwnd is a top-level window.
AW_HIDE	Hides the window. By default, the window is shown.
AW_CENTER	Makes the window appear to collapse inward if AW_HIDE is used or expand outward if the AW_HIDE is not used.
AW_HOR_POSITIVE	Animates the window from left to right. This flag can be used with roll or slide animation. It is ignored when used with AW_CENTER or AW_BLEND.
AW_HOR_NEGATIVE	Animates the window from right to left. This flag can be used with roll or slide animation. It is ignored when used with AW_CENTER or AW_BLEND.
AW_VER_POSITIVE	Animates the window from top to bottom. This flag can be used with roll or slide animation. It is ignored when used with AW_CENTER or AW_BLEND.
AW_VER_NEGATIVE	Animates the window from bottom to top. This flag can be used with roll or slide animation. It is ignored when used with AW_CENTER or AW_BLEND.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. The function will fail in the following situations:

- The window uses the window region.
- The window is already visible and you are trying to show the window.

- The window is already hidden and you are trying to hide the window.

To get extended error information, call the GetLastError function.

Remarks

You can combine AW_HOR_POSITIVE or AW_HOR_NEGATIVE with AW_VER_POSITIVE or AW_VER_NEGATIVE to animate a window diagonally.

The window procedures for the window and its child windows may need to handle any WM_PRINT or WM_PRINTCLIENT messages. Dialog boxes, controls, and common controls already handle WM_PRINTCLIENT. The default window procedure already handles WM_PRINT.

Requirements

Windows NT/2000: Requires Windows 2000 or later.

Windows 95/98: Requires Windows 98 or later.

3.5 AnyPopup

The AnyPopup function indicates whether an owned, visible, top-level pop-up, or overlapped window exists on the screen. The function searches the entire screen, not just the calling application's client area.

Note This function is provided only for compatibility with 16-bit versions of Windows. It is generally not useful.

```
AnyPopup: procedure;
    stdcall;
    returns( "eax" );
    external( "__imp__AnyPopup@0" );
```

Parameters

This function has no parameters.

Return Values

If a pop-up window exists, the return value is nonzero, even if the pop-up window is completely covered by other windows.

If a pop-up window does not exist, the return value is zero.

Remarks

This function does not detect unowned pop-up windows or windows that do not have the WS_VISIBLE style bit set.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.6 AppendMenu

The AppendMenu function appends a new item to the end of the specified menu bar, drop-down menu, submenu, or shortcut menu. You can use this function to specify the content, appearance, and behavior of the menu item.

Note The AppendMenu function has been superseded by the InsertMenuItem function. You can still use Append-

Menu, however, if you do not need any of the extended features of InsertMenuItem.

```
AppendMenu: procedure
(
    hMenu           :dword;
    uFlags          :dword;
    uIDNewItem      :dword;
    lpNewItem       :string
);
    stdcall;
    returns( "eax" );
    external( "__imp__AppendMenuA@16" );
```

Parameters

hMenu

[in] Handle to the menu bar, drop-down menu, submenu, or shortcut menu to be changed.

uFlags

[in] Specifies flags to control the appearance and behavior of the new menu item. This parameter can be a combination of the values listed in the following Remarks section.

uIDNewItem

[in] Specifies either the identifier of the new menu item or, if the uFlags parameter is set to MF_POPUP, a handle to the drop-down menu or submenu.

lpNewItem

[in] Specifies the content of the new menu item. The interpretation of lpNewItem depends on whether the uFlags parameter includes the MF_BITMAP, MF_OWNERDRAW, or MF_STRING flag, as shown in the following table.

Value	Description
MF_BITMAP	Contains a bitmap handle.
MF_OWNERDRAW	Contains an application-supplied value that can be used to maintain additional data related to the menu item. The value is in the item-Data member of the structure pointed to by the lparam parameter of the WM_MEASURE or WM_DRAWITEM message sent when the menu is created or its appearance is updated.
MF_STRING	Contains a pointer to a null-terminated string.

Return Values

If the function succeeds, the return value is nonzero. If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

The application must call the DrawMenuBar function whenever a menu changes, whether or not the menu is in a displayed window.

To get keyboard accelerators to work with bitmap or owner-drawn menu items, the owner of the menu must process the WM_MENUCHAR message. For more information, see Owner-Drawn Menus and the WM_MENUCHAR Message.

The following flags can be set in the uFlags parameter.

Value	Description
MF_BITMAP	Uses a bitmap as the menu item. The lpNewItem parameter contains a handle to the bitmap.

MF_CHECKED	Places a check mark next to the menu item. If the application provides check-mark bitmaps (see SetMenuItemBitmaps, this flag displays the check-mark bitmap next to the menu item.
MF_DISABLED	Disables the menu item so that it cannot be selected, but the flag does not gray it.
MF_ENABLED	Enables the menu item so that it can be selected, and restores it from its grayed state.
MF_GRAYED	Disables the menu item and grays it so that it cannot be selected.
MF_MENUBARBREAK	Functions the same as the MF_MENUBREAK flag for a menu bar. For a drop-down menu, submenu, or shortcut menu, the new column is separated from the old column by a vertical line.
MF_MENUBREAK	Places the item on a new line (for a menu bar) or in a new column (for a drop-down menu, submenu, or shortcut menu) without separating columns.
MF_OWNERDRAW	Specifies that the item is an owner-drawn item. Before the menu is displayed for the first time, the window that owns the menu receives a WM_MEASUREITEM message to retrieve the width and height of the menu item. The WM_DRAWITEM message is then sent to the window procedure of the owner window whenever the appearance of the menu item must be updated.
MF_POPUP	Specifies that the menu item opens a drop-down menu or submenu. The uIDNewItem parameter specifies a handle to the drop-down menu or submenu. This flag is used to add a menu name to a menu bar, or a menu item that opens a submenu to a drop-down menu, submenu, or shortcut menu.
MF_SEPARATOR	Draws a horizontal dividing line. This flag is used only in a drop-down menu, submenu, or shortcut menu. The line cannot be grayed, disabled, or highlighted. The lpNewItem and uIDNewItem parameters are ignored.
MF_STRING	Specifies that the menu item is a text string; the lpNewItem parameter is a pointer to the string.
MF_UNCHECKED	Does not place a check mark next to the item (default). If the application supplies check-mark bitmaps (see SetMenuItemBitmaps), this flag displays the clear bitmap next to the menu item.

The following groups of flags cannot be used together:

- MF_BITMAP, MF_STRING, and MF_OWNERDRAW
- MF_CHECKED and MF_UNCHECKED
- MF_DISABLED, MF_ENABLED, and MF_GRAYED
- MF_MENUBARBREAK and MF_MENUBREAK

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.7 ArrangeIconicWindows

The ArrangeIconicWindows function arranges all the minimized (iconic) child windows of the specified parent window.

```
ArrangeIconicWindows: procedure
(
```

```
    hWnd            :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__ArrangeIconicWindows@4" );
```

Parameters

hWnd

[in] Handle to the parent window.

Return Values

If the function succeeds, the return value is the height of one row of icons.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

An application that maintains its own minimized child windows can use the ArrangeIconicWindows function to arrange icons in a parent window. This function can also arrange icons on the desktop. To retrieve the window handle to the desktop window, use the GetDesktopWindow function.

An application sends the WM_MDIICONARRANGE message to the multiple document interface (MDI) client window to prompt the client window to arrange its minimized MDI child windows.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.8 AttachThreadInput

The AttachThreadInput function attaches the input processing mechanism of one thread to that of another thread.

```
AttachThreadInput: procedure
(
    idAttach        :dword;
    idAttachTo      :dword;
    fAttach         :boolean
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__AttachThreadInput@12" );
```

Parameters

idAttach

[in] Specifies the identifier of the thread to be attached to another thread. The thread to be attached cannot be a system thread.

idAttachTo

[in] Specifies the identifier of the thread to be attached to. This thread cannot be a system thread.

A thread cannot attach to itself. Therefore, idAttachTo cannot equal idAttach.

fAttach

[in] Specifies whether to attach or detach the threads. If this parameter is TRUE, the two threads are attached. If the parameter is FALSE, the threads are detached.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. There is no extended error information; do not call GetLastError.

Remarks

Windows created in different threads typically process input independently of each other. That is, they have their own input states (focus, active, capture windows, key state, queue status, and so on), and they are not synchronized with the input processing of other threads. By using the AttachThreadInput function, a thread can attach its input processing to another thread. This also allows threads to share their input states, so they can call the SetFocus function to set the keyboard focus to a window of a different thread. This also allows threads to get key-state information. These capabilities are not generally possible.

The AttachThreadInput function fails if either of the specified threads does not have a message queue. The system creates a thread's message queue when the thread makes its first call to one of the Win32 USER or GDI functions. The AttachThreadInput function also fails if a journal record hook is installed. Journal record hooks attach all input queues together.

Note that key state, which can be ascertained by calls to the GetKeyState or GetKeyboardState function, is reset after a call to AttachThreadInput.

Windows NT/2000: You cannot attach a thread to a thread in another desktop.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.9 BeginDeferWindowPos

The BeginDeferWindowPos function allocates memory for a multiple-window- position structure and returns the handle to the structure.

```
BeginDeferWindowPos: procedure
(
    nNumWindows     :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__BeginDeferWindowPos@4" );
```

Parameters

nNumWindows

[in] Specifies the initial number of windows for which to store position information. The DeferWindowPos function increases the size of the structure, if necessary.

Return Values

If the function succeeds, the return value identifies the multiple-window – position structure. If insufficient system resources are available to allocate the structure, the return value is NULL. To get extended error information, call GetLastError.

Remarks

The multiple-window-position structure is an internal structure; an application cannot access it directly.

DeferWindowPos fills the multiple-window- position structure with information about the target position for one or more windows about to be moved. The EndDeferWindowPos function accepts the handle to this structure and repositions the windows by using the information stored in the structure.

If any of the windows in the multiple-window- position structure have the SWP_HIDEWINDOW or SWP_SHOWWINDOW flag set, none of the windows are repositioned.

If the system must increase the size of the multiple-window- position structure beyond the initial size specified by the nNumWindows parameter but cannot allocate enough memory to do so, the system fails the entire window positioning sequence (BeginDeferWindowPos, DeferWindowPos, and EndDeferWindowPos). By specifying the maximum size needed, an application can detect and process failure early in the process.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.10 BeginPaint

The BeginPaint function prepares the specified window for painting and fills a PAINTSTRUCT structure with information about the painting.

```
BeginPaint: procedure
(
        hwnd        :dword;
    var lpPaint     :PAINTSTRUCT
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__BeginPaint@8" );
```

Parameters

hwnd

[in] Handle to the window to be repainted.

lpPaint

[out] Pointer to the PAINTSTRUCT structure that will receive painting information.

Return Values

If the function succeeds, the return value is the handle to a display device context for the specified window.

If the function fails, the return value is NULL, indicating that no display device context is available.

Windows NT/ 2000: To get extended error information, call GetLastError.

Remarks

The BeginPaint function automatically sets the clipping region of the device context to exclude any area outside the update region. The update region is set by the InvalidateRect or InvalidateRgn function and by the system after sizing, moving, creating, scrolling, or any other operation that affects the client area. If the update region is marked for erasing, BeginPaint sends a WM_ERASEBKGND message to the window.

An application should not call BeginPaint except in response to a WM_PAINT message. Each call to BeginPaint

must have a corresponding call to the EndPaint function.

If the caret is in the area to be painted, BeginPaint automatically hides the caret to prevent it from being erased.

If the window's class has a background brush, BeginPaint uses that brush to erase the background of the update region before returning.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.11 BlockInput

The BlockInput function blocks keyboard and mouse input events from reaching applications.

```
BlockInput: procedure
(
    fBlockIt        :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__BlockInput@4" );
```

Parameters

fBlockIt

[in] Specifies the function's purpose. If this parameter is TRUE, keyboard and mouse input events are blocked. If this parameter is FALSE, keyboard and mouse events are unblocked. Note that only the thread that blocked input can successfully unblock input.

Return Values

If the function succeeds, the return value is nonzero.

If input is already blocked, the return value is zero. To get extended error information, call GetLastError.

Remarks

When input is blocked, real physical input from the mouse or keyboard will not affect the the input queue's synchronous key state (reported by GetKeyState and GetKeyboardState), nor will it affect the asynchronous key state (reported by GetAsyncKeyState). However, the thread that is blocking input can affect both of these key states by calling SendInput. No other thread can do this.

The system will unblock input in the following cases:

- The thread that blocked input unexpectedly exits without calling BlockInput with fBlock set to FALSE. In this case, the system cleans up properly and re-enables input.
- Windows 95/98: The system displays the Close Program/Fault dialog box. This can occur if the thread faults or if the user presses CTRL+ALT+DEL.
- Windows 2000: The user presses CTRL+ALT+DEL or the system invokes the Hard System Error modal message box (for example, when a program faults or a device fails).

Requirements

Windows NT/2000: Requires Windows 2000 or later.

Windows 95/98: Requires Windows 98 or later.

3.12 BringWindowToTop

The BringWindowToTop function brings the specified window to the top of the Z order. If the window is a top-level window, it is activated. If the window is a child window, the top-level parent window associated with the child window is activated.

```
BringWindowToTop: procedure
(
    hwnd            :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__BringWindowToTop@4" );
```

Parameters

hWnd

[in] Handle to the window to bring to the top of the Z order.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

Use the BringWindowToTop function to uncover any window that is partially or completely obscured by other windows.

Calling this function is similar to calling the SetWindowPos function to change a window's position in the Z order. BringWindowToTop does not make a window a top-level window.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.13 BroadcastSystemMessage

The BroadcastSystemMessage function sends a message to the specified recipients. The recipients can be applications, installable drivers, network drivers, system-level device drivers, or any combination of these system components.

To receive additional information if the request is defined, use the BroadcastSystemMessageEx function.

```
BroadcastSystemMessage: procedure
(
        dwFlags         :dword;
    var lpdwRecipients  :dword;
        uiMessage       :dword;
        _wParam         :dword;
        _lParam         :dword
```

```
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__BroadcastSystemMessage@20" );
```

Parameters

dwFlags

[in] Specifies the broadcast option. This parameter can be one or more of the following values.

Value	Meaning
BSF_ALLOWSFW	Windows 2000: Enables the recipient to set the foreground window while processing the message.
BSF_FLUSHDISK	Flushes the disk after each recipient processes the message.
BSF_FORCEIFHUNG	Continues to broadcast the message, even if the time-out period elapses or one of the recipients is hung.
BSF_IGNORECURRENTTASK	Does not send the message to windows that belong to the current task. This prevents an application from receiving its own message.
BSF_NOHANG	Forces a hung application to time out. If one of the recipients times out, do not continue broadcasting the message.
BSF_NOTIMEOUTIFNOTHUNG	Waits for a response to the message, as long as the recipient is not hung. Do not time out.
BSF_POSTMESSAGE	Posts the message. Do not use in combination with BSF_QUERY.
BSF_QUERY	Sends the message to one recipient at a time, sending to a subsequent recipient only if the current recipient returns TRUE.
BSF_SENDNOTIFYMESSAGE	Windows 2000: Sends the message using SendNotifyMessage function. Do not use in combination with BSF_QUERY.

lpdwRecipients

[in] Pointer to a variable that contains and receives information about the recipients of the message. This parameter can be one or more of the following values.

Value	Meaning
BSM_ALLCOMPONENTS	Broadcast to all system components.
BSM_ALLDESKTOPS	Windows NT/2000: Broadcast to all desktops. Requires the SE_TCB_NAME privilege.
BSM_APPLICATIONS	Broadcast to applications.
BSM_INSTALLABLEDRIVERS	Windows 95/98: Broadcast to installable drivers.
BSM_NETDRIVER	Windows 95/98: Broadcast to network drivers.
BSM_VXDS	Windows 95/98: Broadcast to all system-level device drivers.

When the function returns, this variable receives a combination of these values identifying which recipients actually received the message.

If this parameter is NULL, the function broadcasts to all components.

uiMessage

[in] Specifies the message to be sent.

wParam

[in] Specifies additional message-specific information.

lParam

[in] Specifies additional message-specific information.

Return Values

If the function succeeds, the return value is a positive value.

If the function is unable to broadcast the message, the return value is –1.

If the dwFlags parameter is BSF_QUERY and at least one recipient returned BROADCAST_QUERY_DENY to the corresponding message, the return value is zero. To get extended error information, call GetLastError.

Remarks

If BSF_QUERY is not specified, the function sends the specified message to all requested recipients, ignoring values returned by those recipients.

Requirements

Windows NT/2000: Requires Windows NT 4.0 or later.

Windows 95/98: Requires Windows 95 or later.

3.14 CallMsgFilter

The CallMsgFilter function passes the specified message and hook code to the hook procedures associated with the WH_SYSMSGFILTER and WH_MSGFILTER hooks. A WH_SYSMSGFILTER or WH_MSGFILTER hook procedure is an application-defined callback function that examines and, optionally, modifies messages for a dialog box, message box, menu, or scroll bar.

```
CallMsgFilter: procedure
(
    var lpMsg           :MSG;
        nCode           :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__CallMsgFilter@8" );
```

Parameters

lpMsg

[in] Pointer to an MSG structure that contains the message to be passed to the hook procedures.

nCode

[in] Specifies an application-defined code used by the hook procedure to determine how to process the message. The code must not have the same value as system-defined hook codes (MSGF_ and HC_) associated with the WH_SYSMSGFILTER and WH_MSGFILTER hooks.

Return Values

If the application should process the message further, the return value is zero.

If the application should not process the message further, the return value is nonzero.

Remarks

The system calls CallMsgFilter to enable applications to examine and control the flow of messages during internal processing of dialog boxes, message boxes, menus, and scroll bars, or when the user activates a different window by pressing the ALT+TAB key combination.

Install this hook procedure by using the SetWindowsHookEx function.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.15 CallNextHookEx

The CallNextHookEx function passes the hook information to the next hook procedure in the current hook chain. A hook procedure can call this function either before or after processing the hook information.

```
CallNextHookEx: procedure
(
    hhk             :dword;
    nCode           :dword;
    _wParam         :dword;
    _lParam         :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__CallNextHookEx@16" );
```

Parameters

hhk

[in] Handle to the current hook. An application receives this handle as a result of a previous call to the SetWindowsHookEx function.

nCode

[in] Specifies the hook code passed to the current hook procedure. The next hook procedure uses this code to determine how to process the hook information.

wParam

[in] Specifies the wParam value passed to the current hook procedure. The meaning of this parameter depends on the type of hook associated with the current hook chain.

lParam

[in] Specifies the lParam value passed to the current hook procedure. The meaning of this parameter depends on the type of hook associated with the current hook chain.

Return Values

The return value is the value returned by the next hook procedure in the chain. The current hook procedure must also return this value. The meaning of the return value depends on the hook type. For more information, see the descriptions of the individual hook procedures.

Remarks

Hook procedures are installed in chains for particular hook types. CallNextHookEx calls the next hook in the chain.

Calling CallNextHookEx is optional, but it is highly recommended; otherwise, other applications that have installed hooks will not receive hook notifications and may behave incorrectly as a result. You should call CallNextHookEx unless you absolutely need to prevent the notification from being seen by other applications.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.16 CallWindowProc

The CallWindowProc function passes message information to the specified window procedure.

```
CallWindowProc: procedure
(
    lpPrevWndFunc   :WNDPROC;
    hWnd            :dword;
    Msg             :dword;
    _wParam         :dword;
    _lParam         :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__CallWindowProcA@20" );
```

Parameters

lpPrevWndFunc

[in] Pointer to the previous window procedure.

If this value is obtained by calling the GetWindowLong function with the nIndex parameter set to GWL_WNDPROC or DWL_DLGPROC, it is actually either the address of a window or dialog box procedure, or a special internal value meaningful only to CallWindowProc.

hWnd

[in] Handle to the window procedure to receive the message.

Msg

[in] Specifies the message.

wParam

[in] Specifies additional message-specific information. The contents of this parameter depend on the value of the Msg parameter.

lParam

[in] Specifies additional message-specific information. The contents of this parameter depend on the value of the Msg parameter.

Return Values

The return value specifies the result of the message processing and depends on the message sent.

Remarks

Use the CallWindowProc function for window subclassing. Usually, all windows with the same class share one window procedure. A subclass is a window or set of windows with the same class whose messages are intercepted and processed by another window procedure (or procedures) before being passed to the window procedure of the class.

The SetWindowLong function creates the subclass by changing the window procedure associated with a particular window, causing the system to call the new window procedure instead of the previous one. An application must pass any messages not processed by the new window procedure to the previous window procedure by call-

ing CallWindowProc. This allows the application to create a chain of window procedures.

If STRICT is defined, the lpPrevWndFunc parameter has the data type WNDPROC. The WNDPROC type is declared as follows:

LRESULT (CALLBACK* WNDPROC) (HWND, UINT, WPARAM, LPARAM);

If STRICT is not defined, the lpPrevWndFunc parameter has the data type FARPROC. The FARPROC type is declared as follows:

int (FAR WINAPI * FARPROC) ()

In C, the FARPROC declaration indicates a callback function that has an unspecified parameter list. In C++, however, the empty parameter list in the declaration indicates that a function has no parameters. This subtle distinction can break careless code. Following is one way to handle this situation:

```
#ifdef STRICT
  WNDPROC MyWindowProcedure
#else
  FARPROC MyWindowProcedure
#endif
...
   lResult = CallWindowProc(MyWindowProcedure, ...) ;
```

For further information about functions declared with empty argument lists, refer to The C++ Programming Language, Second Edition, by Bjarne Stroustrup.

Windows NT/2000: The CallWindowProc function handles Unicode-to-ANSI conversion. You cannot take advantage of this conversion if you call the window procedure directly.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.17 CascadeWindows

The CascadeWindows function cascades the specified child windows of the specified parent window.

```
CascadeWindows: procedure
(
        hwndParent  :dword;
        wHow        :dword;
    var lpRect      :RECT;
        cKids       :dword;
    var lpKids      :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__CascadeWindows@20" );
```

Parameters

hwndParent

[in] Handle to the parent window. If this parameter is NULL, the desktop window is assumed.

wHow

[in] Specifies a cascade flag. This parameter can be one or more of the following values.

Value	Meaning
MDITILE_SKIPDISABLED	Prevents disabled MDI child windows from being cascaded.
MDITILE_ZORDER	Windows 2000: Arranges the windows in Z order. If this value is not specified, the windows are arranged using the order specified in the lpKids array.

lpRect

[in] Pointer to a RECT structure that specifies the rectangular area, in client coordinates, within which the windows are arranged. This parameter can be NULL, in which case the client area of the parent window is used.

cKids

[in] Specifies the number of elements in the array specified by the lpKids parameter. This parameter is ignored if lpKids is NULL.

lpKids

[in] Pointer to an array of handles to the child windows to arrange. If this parameter is NULL, all child windows of the specified parent window (or of the desktop window) are arranged.

Return Values

If the function succeeds, the return value is the number of windows arranged.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

By default, CascadeWindows arranges the windows in the order provided by the lpKids array, but preserves the Z order. If you specify the MDITILE_ZORDER flag, CascadeWindows arranges the windows in Z order.

Calling CascadeWindows causes all maximized windows to be restored to their previous size.

Requirements

Windows NT/2000: Requires Windows NT 4.0 or later.

Windows 95/98: Requires Windows 95 or later.

3.18 ChangeClipboardChain

The ChangeClipboardChain function removes a specified window from the chain of clipboard viewers.

```
ChangeClipboardChain: procedure
(
    hWndRemove      :dword;
    hWndNewNext     :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__ChangeClipboardChain@8" );
```

Parameters

hWndRemove

[in] Handle to the window to be removed from the chain. The handle must have been passed to the SetClipboard-Viewer function.

hWndNewNext

[in] Handle to the window that follows the hWndRemove window in the clipboard viewer chain. (This is the handle returned by SetClipboardViewer, unless the sequence was changed in response to a WM_CHANGECBCHAIN message.)

Return Values

The return value indicates the result of passing the WM_CHANGECBCHAIN message to the windows in the clipboard viewer chain. Because a window in the chain typically returns FALSE when it processes WM_CHANGECBCHAIN, the return value from ChangeClipboardChain is typically FALSE. If there is only one window in the chain, the return value is typically TRUE.

Remarks

The window identified by hWndNewNext replaces the hWndRemove window in the chain. The SetClipboard-Viewer function sends a WM_CHANGECBCHAIN message to the first window in the clipboard viewer chain.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.19 ChangeDisplaySettings

The ChangeDisplaySettings function changes the settings of the default display device to the specified graphics mode.

To change the settings of a specified display device, use the ChangeDisplaySettingsEx function.

```
ChangeDisplaySettings: procedure
(
    var lpDevMode   :DEVMODE;
        dwFlags     :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__ChangeDisplaySettingsA@8" );
```

Parameters

lpDevMode

[in] Pointer to a DEVMODE structure that describes the new graphics mode. If lpDevMode is NULL, all the values currently in the registry will be used for the display setting. Passing NULL for the lpDevMode parameter and 0 for the dwFlags parameter is the easiest way to return to the default mode after a dynamic mode change.

The dmSize member of DEVMODE must be initialized to the size, in bytes, of the DEVMODE structure. The dmDriverExtra member of DEVMODE must be initialized to indicate the number of bytes of private driver data following the DEVMODE structure. In addition, you can use any or all of the following members

of the DEVMODE structure.

Member	Meaning
dmBitsPerPel	Bits per pixel
dmPelsWidth	Pixel width
dmPelsHeight	Pixel height
dmDisplayFlags	Mode flags
dmDisplayFrequency	Mode frequency
dmPosition	Windows 98, Windows 2000: Position of the device in a multimonitor configuration

In addition to using one or more of the preceding DEVMODE members, you must also set one or more of the following values in the dmFields member to change the display setting.

Value	Meaning
DM_BITSPERPEL	Use the dmBitsPerPel value.
DM_PELSWIDTH	Use the dmPelsWidth value.
DM_PELSHEIGHT	Use the dmPelsHeight value.
DM_DISPLAYFLAGS	Use the dmDisplayFlags value.
DM_DISPLAYFREQUENCY	Use the dmDisplayFrequency value.
DM_POSITION	Windows 98, Windows 2000: Use the dmPosition value.

dwflags

[in] Indicates how the graphics mode should be changed. This parameter can be one of the following values.

Value	Meaning
0	The graphics mode for the current screen will be changed dynamically.
CDS_UPDATEREGISTRY	The graphics mode for the current screen will be changed dynamically and the graphics mode will be updated in the registry. The mode information is stored in the USER profile.
CDS_TEST	The system tests if the requested graphics mode could be set.
CDS_FULLSCREEN	The mode is temporary in nature. Windows NT/2000: If you change to and from another desktop, this mode will not be reset.
CDS_GLOBAL	The settings will be saved in the global settings area so that they will affect all users on the machine. Otherwise, only the settings for the user are modified. This flag is only valid when specified with the CDS_UPDATEREGISTRY flag.
CDS_SET_PRIMARY	This device will become the primary device.
CDS_RESET	The settings should be changed, even if the requested settings are the same as the current settings.
CDS_NORESET	The settings will be saved in the registry, but will not take affect. This flag is only valid when specified with the CDS_UPDATEREGISTRY flag.

Specifying CDS_TEST allows an application to determine which graphics modes are actually valid, without causing the system to change to that graphics mode.

If CDS_UPDATEREGISTRY is specified and it is possible to change the graphics mode dynamically, the information is stored in the registry and DISP_CHANGE_SUCCESSFUL is returned. If it is not possible to change the graphics mode dynamically, the information is stored in the registry and DISP_CHANGE_RESTART is returned.

Windows NT/2000: If CDS_UPDATEREGISTRY is specified and the information could not be stored in the registry, the graphics mode is not changed and DISP_CHANGE_NOTUPDATED is returned.

Return Values

The ChangeDisplaySettings function returns one of the following values.

Value	Meaning
DISP_CHANGE_SUCCESSFUL	The settings change was successful.
DISP_CHANGE_RESTART	The computer must be restarted in order for the graphics mode to work.
DISP_CHANGE_BADFLAGS	An invalid set of flags was passed in.
DISP_CHANGE_BADPARAM	An invalid parameter was passed in. This can include an invalid flag or combination of flags.
DISP_CHANGE_FAILED	The display driver failed the specified graphics mode.
DISP_CHANGE_BADMODE	The graphics mode is not supported.
DISP_CHANGE_NOTUPDATED	Windows NT/2000: Unable to write settings to the registry.

Remarks

To ensure that the DEVMODE structure passed to ChangeDisplaySettings is valid and contains only values supported by the display driver, use the DEVMODE returned by the EnumDisplaySettings function.

When the display mode is changed dynamically, the WM_DISPLAYCHANGE message is sent to all running applications with the following message parameters.

Parameters	Meaning
wParam	New bits per pixel
LOWORD(lParam)	New pixel width
HIWORD(lParam)	New pixel height

Windows 95: If the calling thread has any top-level windows, ChangeDisplaySettings sends these windows the WM_DISPLAYCHANGE message right away (for all other windows the message is posted). This may cause the shell to get its message too soon and could squash icons. To avoid this problem, have ChangeDisplaySettings do resolution switching by calling on a thread with no windows, for example, a new thread.

Requirements

Windows NT/2000: Requires Windows NT 3.5 or later.

Windows 95/98: Requires Windows 95 or later.

3.20 ChangeDisplaySettingsEx

The ChangeDisplaySettingsEx function changes the settings of the specified display device to the specified graphics mode.

```
ChangeDisplaySettingsEx: procedure
(
        lpszDeviceName  :string;
    var lpDevMode       :DEVMODE;
        hwnd            :dword;
        dwflags         :dword;
    var _lParam         :var
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__ChangeDisplaySettingsExA@20" );
```

Parameters

lpszDeviceName

[in] Pointer to a null-terminated string that specifies the display device whose graphics mode the function will obtain information about. Only display device names as returned by EnumDisplayDevices are valid. See EnumDisplayDevices for further information on the names associated with these display devices.

The lpszDeviceName parameter can be NULL. A NULL value specifies the default display device. The default device can be determined by calling EnumDisplayDevices and checking for the DISPLAY_DEVICE_PRIMARY_DEVICE flag.

lpDevMode

[in] Pointer to a DEVMODE structure that describes the new graphics mode. If lpDevMode is NULL, all the values currently in the registry will be used for the display setting. Passing NULL for the lpDevMode parameter and 0 for the dwFlags parameter is the easiest way to return to the default mode after a dynamic mode change.

The dmSize member must be initialized to the size, in bytes, of the DEVMODE structure. The dmDriverExtra member must be initialized to indicate the number of bytes of private driver data following the DEVMODE structure. In addition, you can use any of the following members of the DEVMODE structure.

Member	Meaning
dmBitsPerPel	Bits per pixel
dmPelsWidth	Pixel width
dmPelsHeight	Pixel height
dmDisplayFlags	Mode flags
dmDisplayFrequency	Mode frequency
dmPosition	Windows 98, Windows 2000: Position of the device in a multi-monitor configuration.

In addition to using one or more of the preceding DEVMODE members, you must also set one or more of the following values in the dmFields member to change the display settings.

Value	Meaning
DM_BITSPERPEL	Use the dmBitsPerPel value.
DM_PELSWIDTH	Use the dmPelsWidth value.
DM_PELSHEIGHT	Use the dmPelsHeight value.
DM_DISPLAYFLAGS	Use the dmDisplayFlags value.
DM_DISPLAYFREQUENCY	Use the dmDisplayFrequency value.
DM_POSITION	Windows 98, Windows 2000: Use the dmPosition value.

hwnd

Reserved; must be NULL.

dwflags

[in] Indicates how the graphics mode should be changed. This parameter can be one of the following values.

Value	Meaning
0	The graphics mode for the current screen will be changed dynamically.
CDS_FULLSCREEN	The mode is temporary in nature. Windows NT/2000: If you change to and from another desktop, this mode will not be reset.
CDS_GLOBAL	The settings will be saved in the global settings area so that they will affect all users on the machine. Otherwise, only the settings for the user are modified. This flag is only valid when specified with the CDS_UPDATEREGISTRY flag.

CDS_NORESET	The settings will be saved in the registry, but will not take effect. This flag is only valid when specified with the CDS_UPDATEREGISTRY flag.
CDS_RESET	The settings should be changed, even if the requested settings are the same as the current settings.
CDS_SET_PRIMARY	This device will become the primary device.
CDS_TEST	The system tests if the requested graphics mode could be set.
CDS_UPDATEREGISTRY	The graphics mode for the current screen will be changed dynamically and the graphics mode will be updated in the registry. The mode information is stored in the USER profile.
CDS_VIDEOPARAMETERS	Windows NT/2000: When set, the lParam parameter is a pointer to a VIDEOPARAMETERS structure.

Specifying CDS_TEST allows an application to determine which graphics modes are actually valid, without causing the system to change to them.

If CDS_UPDATEREGISTRY is specified and it is possible to change the graphics mode dynamically, the information is stored in the registry and DISP_CHANGE_SUCCESSFUL is returned. If it is not possible to change the graphics mode dynamically, the information is stored in the registry and DISP_CHANGE_RESTART is returned.

Windows NT/2000: If CDS_UPDATEREGISTRY is specified and the information could not be stored in the registry, the graphics mode is not changed and DISP_CHANGE_NOTUPDATED is returned.

lParam

Windows NT/2000: [in] If dwFlags is CDS_VIDEOPARAMETERS, lParam is a pointer to a VIDEOPARAMETERS structure. Otherwise lParam must be NULL.

Return Values

The ChangeDisplaySettingsEx function returns one of the following values.

Value	Meaning
DISP_CHANGE_BADFLAGS	An invalid set of flags was passed in.
DISP_CHANGE_BADMODE	The graphics mode is not supported.
DISP_CHANGE_BADPARAM	An invalid parameter was passed in. This can include an invalid flag or combination of flags.
DISP_CHANGE_FAILED	The display driver failed the specified graphics mode.
DISP_CHANGE_NOTUPDATED	Windows NT/2000: Unable to write settings to the registry.
DISP_CHANGE_RESTART	The computer must be restarted for the graphics mode to work.
DISP_CHANGE_SUCCESSFUL	The settings change was successful.

Remarks

To ensure that the DEVMODE structure passed to ChangeDisplaySettingsEx is valid and contains only values supported by the display driver, use the DEVMODE returned by the EnumDisplaySettings function.

When adding a display monitor to a multiple-monitor system programmatically, set DEVMODE.dmFields to DM_POSITION and specify a position (in DEVMODE.dmPosition) for the monitor you are adding that is adjacent to at least one pixel of the display area of an existing monitor. To detach the monitor, set DEVMODE.dmFields to DM_POSITION but set DEVMODE.dmPelsWidth and DEVMODE.dmPelsHeight to zero. For more information, see Multiple Display Monitors.

When the display mode is changed dynamically, the WM_DISPLAYCHANGE message is sent to all running applications with the following message parameters.

Parameters	Meaning
wParam	New bits per pixel
LOWORD(lParam)	New pixel width
HIWORD(lParam)	New pixel height

Requirements

Windows NT/2000: Requires Windows 2000 or later.

Windows 95/98: Requires Windows 98 or later.

3.21 CharLower

The CharLower function converts a character string or a single character to lowercase. If the operand is a character string, the function converts the characters in place.

```
CharLower: procedure
(
    lpsz        :string
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__CharLowerA@4" );
```

Parameters

lpsz

[in/out] Pointer to a null-terminated string or specifies a single character. If the high-order word of this parameter is zero, the low-order word must contain a single character to be converted.

Return Values

If the operand is a character string, the function returns a pointer to the converted string. Since the string is converted in place, the return value is equal to lpsz.

If the operand is a single character, the return value is a 32-bit value whose high-order word is zero, and low-order word contains the converted character.

There is no indication of success or failure. Failure is rare. There is no extended error information for this function; do not call GetLastError.

Remarks

Windows NT/ 2000: To make the conversion, the function uses the language driver for the current language selected by the user at setup or by using Control Panel. If no language has been selected, the system completes the conversion by using internal default mapping. The conversion is made based on the code page associated with the process locale.

Windows 95: The function makes the conversion based on the information associated with the user's default locale, which is the locale selected by the user at setup or by using Control Panel. Windows 95 does not have language drivers.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.22 CharLowerBuff

The CharLowerBuff function converts uppercase characters in a buffer to lowercase characters. The function converts the characters in place.

```
CharLowerBuff: procedure
(
    lpsz        :string;
    cchLength   :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__CharLowerBuffA@8" );
```

Parameters

lpsz

[in/out] Pointer to a buffer containing one or more characters to process.

cchLength

[in] Specifies the size, in TCHARs, of the buffer pointed to by lpsz. This refers to bytes for ANSI versions of the function or characters for Unicode versions. The function examines each character, and converts uppercase characters to lowercase characters. The function examines the number of characters indicated by cchLength, even if one or more characters are null characters.

Return Values

The return value is the number of TCHARs processed. For example, if CharLowerBuff("Acme of Operating Systems", 10) succeeds, the return value is 10.

Remarks

Windows NT/ 2000: To make the conversion, the function uses the language driver for the current language selected by the user at setup or by using Control Panel. If no language has been selected, the system completes the conversion by using internal default mapping. The conversion is made based on the code page associated with the process locale.

Windows 95: The function makes the conversion based on the information associated with the user's default locale, which is the locale selected by the user at setup or by using Control Panel. Windows 95 does not have language drivers.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.23 CharNext

The CharNext function retrieves a pointer to the next character in a string.

```
CharNext: procedure
(
    lpsz        :string
);
    @stdcall;
    @returns( "eax" );
```

```
    @external( "__imp__CharNextA@4" );
```

Parameters

lpsz

[in] Pointer to a character in a null-terminated string.

Return Values

The return value is a pointer to the next character in the string, or to the terminating null character if at the end of the string.

If lpsz points to the terminating null character, the return value is equal to lpsz.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.24 CharPrev

The CharPrev function retrieves a pointer to the preceding character in a string.

```
CharPrev: procedure
(
    lpszStart   :string;
    lpszCurrent :string
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__CharPrevA@8" );
```

Parameters

lpszStart

[in] Pointer to the beginning of the string.

lpszCurrent

[in] Pointer to a character in a null-terminated string.

Return Values

The return value is a pointer to the preceding character in the string, or to the first character in the string if the lpszCurrent parameter equals the lpszStart parameter.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.25 CharToOem

The CharToOem function translates a string into the OEM-defined character set.

```
CharToOem: procedure
(
    lpszSrc     :string;
    lpszDest    :string
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__CharToOemA@8" );
```

Parameters

lpszSrc

[in] Pointer to the null-terminated string to translate.

lpszDst

[out] Pointer to the buffer for the translated string. If the CharToOem function is being used as an ANSI function, the string can be translated in place by setting the lpszDst parameter to the same address as the lpszSrc parameter. This cannot be done if CharToOem is being used as a wide-character function.

Return Values

The return value is always nonzero.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.26 CharToOemBuff

The CharToOemBuff function translates a specified number of characters in a string into the OEM-defined character set.

```
CharToOemBuff: procedure
(
    lpszSrc         :string;
    lpszDst         :string;
    cchDstLength    :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__CharToOemBuffA@12" );
```

Parameters

lpszSrc

[in] Pointer to the null-terminated string to translate.

lpszDst

[out] Pointer to the buffer for the translated string. If the CharToOemBuff function is being used as an ANSI function, the string can be translated in place by setting the lpszDst parameter to the same address as the lpszSrc parameter. This cannot be done if CharToOemBuff is being used as a wide-character function.

cchDstLength

[in] Specifies the number of TCHARs to translate in the string identified by the lpszSrc parameter. This refers

to bytes for ANSI versions of the function or characters for Unicode versions.

Return Values

The return value is always nonzero.

Remarks

Unlike the CharToOem function, the CharToOemBuff function does not stop converting characters when it encounters a null character in the buffer pointed to by lpszSrc. The CharToOemBuff function converts all cchDstLength characters.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.27 CharUpper

The CharUpper function converts a character string or a single character to uppercase. If the operand is a character string, the function converts the characters in place.

```
CharUpper: procedure
(
    lpsz            :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__CharUpperA@4" );
```

Parameters

lpsz

[in/out] Pointer to a null-terminated string or specifies a single character. If the high-order word of this parameter is zero, the low-order word must contain a single character to be converted.

Return Values

If the operand is a character string, the function returns a pointer to the converted string. Since the string is converted in place, the return value is equal to lpsz.

If the operand is a single character, the return value is a 32-bit value whose high-order word is zero, and low-order word contains the converted character.

There is no indication of success or failure. Failure is rare. There is no extended error information for this function; do no call GetLastError.

Remarks

Windows NT/ 2000: To make the conversion, the function uses the language driver for the current language selected by the user at setup or by using Control Panel. If no language has been selected, the system completes the conversion by using internal default mapping. The conversion is made based on the code page associated with the process locale.

Windows 95: The function makes the conversion based on the information associated with the user's default locale, which is the locale selected by the user at setup or by using Control Panel. Windows 95 does not have language drivers.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.28 CharUpperBuff

The CharUpperBuff function converts lowercase characters in a buffer to uppercase characters. The function converts the characters in place.

```
CharUpperBuff: procedure
(
    lpsz            :string;
    cchLength       :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__CharUpperBuffA@8" );
```

Parameters

lpsz

[in] Pointer to a buffer containing one or more characters to process.

cchLength

[in] Specifies the size, in TCHARs, of the buffer pointed to by lpsz. This refers to bytes for ANSI versions of the function or characters for Unicode versions.

The function examines each character, and converts lowercase characters to uppercase characters. The function examines the number of characters indicated by cchLength, even if one or more characters are null characters.

Return Values

The return value is the number of TCHARs processed.

For example, if CharUpperBuff("Zenith of API Sets", 10) succeeds, the return value is 10.

Remarks

Windows NT/ 2000: To make the conversion, the function uses the language driver for the current language selected by the user at setup or by using Control Panel. If no language has been selected, the system completes the conversion by using internal default mapping. The conversion is made based on the code page associated with the process locale.

Windows 95: The function makes the conversion based on the information associated with the user's default locale, which is the locale selected by the user at setup or by using Control Panel. Windows 95 does not have language drivers.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.29 CheckDlgButton

The CheckDlgButton function changes the check state of a button control.

```
CheckDlgButton: procedure
(
    hDlg            :dword;
    nIDButton       :dword;
    uCheck          :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__CheckDlgButton@12" );
```

Parameters

hDlg

[in] Handle to the dialog box that contains the button.

nIDButton

[in] Specifies the identifier of the button to modify.

uCheck

[in] Specifies the check state of the button. This parameter can be one of the following values.

Value	Meaning
BST_CHECKED	Sets the button state to checked.
BST_INDETERMINATE	Sets the button state to grayed, indicating an indeterminate state. Use this value only if the button has the BS_3STATE or BS_AUTO3STATE style.
BST_UNCHECKED	Sets the button state to cleared

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

The CheckDlgButton function sends a BM_SETCHECK message to the specified button control in the specified dialog box.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.30 CheckMenuItem

The CheckMenuItem function sets the state of the specified menu item's check-mark attribute to either selected or clear.

Note The CheckMenuItem function has been superseded by the SetMenuItemInfo function. You can still use CheckMenuItem, however, if you do not need any of the extended features of SetMenuItemInfo.

```
CheckMenuItem: procedure
(
    hmenu           :dword;
    uIDCheckItem    :dword;
    uCheck          :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__CheckMenuItem@12" );
```

Parameters

hmenu

[in] Handle to the menu of interest.

uIDCheckItem

[in] Specifies the menu item whose check-mark attribute is to be set, as determined by the uCheck parameter.

uCheck

[in] Specifies flags that control the interpretation of the uIDCheckItem parameter and the state of the menu item's check-mark attribute. This parameter can be a combination of either MF_BYCOMMAND, or MF_BYPOSITION and MF_CHECKED or MF_UNCHECKED.

Value	Meaning
MF_BYCOMMAND	Indicates that the uIDCheckItem parameter gives the identifier of the menu item. The MF_BYCOMMAND flag is the default, if neither the MF_BYCOMMAND nor MF_BYPOSITION flag is specified.
MF_BYPOSITION	Indicates that the uIDCheckItem parameter gives the zero-based relative position of the menu item.
MF_CHECKED	Sets the check-mark attribute to the selected state.
MF_UNCHECKED	Sets the check-mark attribute to the clear state.

Return Values

The return value specifies the previous state of the menu item (either MF_CHECKED or MF_UNCHECKED). If the menu item does not exist, the return value is -1.

Remarks

An item in a menu bar cannot have a check mark.

The uIDCheckItem parameter identifies a item that opens a submenu or a command item. For a item that opens a submenu, the uIDCheckItem parameter must specify the position of the item. For a command item, the uIDCheckItem parameter can specify either the item's position or its identifier.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.31 CheckMenuRadioItem

The CheckMenuRadioItem function checks a specified menu item and makes it a radio item. At the same time, the function clears all other menu items in the associated group and clears the radio-item type flag for those items.

```
CheckMenuRadioItem: procedure
```

```
(
    hmenu           :dword;
    idFirst         :dword;
    idLast          :dword;
    idCheck         :dword;
    uFlags          :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__CheckMenuRadioItem@20" );
```

Parameters

hmenu

[in] Handle to the menu that contains the group of menu items.

idFirst

[in] Identifier or position of the first menu item in the group.

idLast

[in] Identifier or position of the last menu item in the group.

idCheck

[in] Identifier or position of the menu item to check.

uFlags

[in] Value specifying the meaning of idFirst, idLast, and idCheck. If this parameter is MF_BYCOMMAND, the other parameters specify menu item identifiers. If it is MF_BYPOSITION, the other parameters specify the menu item positions.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, use the GetLastError function.

Remarks

The CheckMenuRadioItem function sets the MFT_RADIOCHECK type flag and the MFS_CHECKED state for the item specified by idCheck and, at the same time, clears both flags for all other items in the group. The selected item is displayed using a bullet bitmap instead of a check-mark bitmap.

For more information about menu item type and state flags, see the MENUITEMINFO structure.

Requirements

Windows NT/2000: Requires Windows NT 4.0 or later.

Windows 95/98: Requires Windows 95 or later.

3.32 CheckRadioButton

The CheckRadioButton function adds a check mark to (checks) a specified radio button in a group and removes a check mark from (clears) all other radio buttons in the group.

```
CheckRadioButton: procedure
(
    hDlg            :dword;
```

```
    nIDFirstButton  :dword;
    nIDLastButton   :dword;
    nIDCheckButton  :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__CheckRadioButton@16" );
```

Parameters

hDlg

[in] Handle to the dialog box that contains the radio button.

nIDFirstButton

[in] Specifies the identifier of the first radio button in the group.

nIDLastButton

[in] Specifies the identifier of the last radio button in the group.

nIDCheckButton

[in] Specifies the identifier of the radio button to select.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

The CheckRadioButton function sends a BM_SETCHECK message to each of the radio buttons in the indicated group.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.33 ChildWindowFromPoint

The ChildWindowFromPoint function determines which, if any, of the child windows belonging to a parent window contains the specified point.

To skip certain child windows, use the ChildWindowFromPointEx function.

```
ChildWindowFromPoint: procedure
(
    hWndParent      :dword;
    _point          :POINT
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__ChildWindowFromPoint@12" );
```

Parameters

hWndParent

[in] Handle to the parent window.

Point

[in] Specifies a POINT structure that defines the client coordinates of the point to be checked.

Return Values

The return value is a handle to the child window that contains the point, even if the child window is hidden or disabled. If the point lies outside the parent window, the return value is NULL. If the point is within the parent window but not within any child window, the return value is a handle to the parent window.

Remarks

The system maintains an internal list, containing the handles of the child windows associated with a parent window. The order of the handles in the list depends on the Z order of the child windows. If more than one child window contains the specified point, the system returns a handle to the first window in the list that contains the point.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.34 ChildWindowFromPointEx

The ChildWindowFromPointEx function determines which, if any, of the child windows belonging to the specified parent window contains the specified point. The function can ignore invisible, disabled, and transparent child windows.

```
ChildWindowFromPointEx: procedure
(
    hwndParent      :dword;
    pt              :POINT;
    uFlags          :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__ChildWindowFromPointEx@16" );
```

Parameters

hwndParent

[in] Handle to the parent window.

pt

[in] Specifies a POINT structure that defines the client coordinates of the point to be checked.

uFlags

[in] Specifies which child windows to skip. This parameter can be one or more of the following values.

Value	Meaning
CWP_ALL	Does not skip any child windows
CWP_SKIPINVISIBLE	Skips invisible child windows
CWP_SKIPDISABLED	Skips disabled child windows
CWP_SKIPTRANSPARENT	Skips transparent child windows

Return Values

The return value is a handle to the first child window that contains the point and meets the criteria specified by

uFlags. If the point is within the parent window but not within any child window that meets the criteria, the return value is a handle to the parent window. If the point lies outside the parent window or if the function fails, the return value is NULL.

Remarks

The system maintains an internal list that contains the handles of the child windows associated with a parent window. The order of the handles in the list depends on the Z order of the child windows. If more than one child window contains the specified point, the system returns a handle to the first window in the list that contains the point and meets the criteria specified by uFlags.

Requirements

Windows NT/2000: Requires Windows NT 4.0 or later.

Windows 95/98: Requires Windows 95 or later.

3.35 ClientToScreen

The ClientToScreen function converts the client-area coordinates of a specified point to screen coordinates.

```
ClientToScreen: procedure
(
        hWnd        :dword;
    var lpPoint     :POINT
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__ClientToScreen@8" );
```

Parameters

hWnd

[in] Handle to the window whose client area is used for the conversion.

lpPoint

[in/out] Pointer to a POINT structure that contains the client coordinates to be converted. The new screen coordinates are copied into this structure if the function succeeds.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Windows NT/ 2000: To get extended error information, call GetLastError.

Remarks

The ClientToScreen function replaces the client-area coordinates in the POINT structure with the screen coordinates. The screen coordinates are relative to the upper-left corner of the screen. Note, a screen-coordinate point that is above the window's client area has a negative y-coordinate. Similarly, a screen coordinate to the left of a client area has a negative x-coordinate.

All coordinates are device coordinates.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.36 ClipCursor

The ClipCursor function confines the cursor to a rectangular area on the screen. If a subsequent cursor position (set by the SetCursorPos function or the mouse) lies outside the rectangle, the system automatically adjusts the position to keep the cursor inside the rectangular area.

```
ClipCursor: procedure
(
    var lpRect      :RECT
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__ClipCursor@4" );
```

Parameters

lpRect

[in] Pointer to the RECT structure that contains the screen coordinates of the upper-left and lower-right corners of the confining rectangle. If this parameter is NULL, the cursor is free to move anywhere on the screen.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

The cursor is a shared resource. If an application confines the cursor, it must release the cursor by using ClipCursor before relinquishing control to another application.

The calling process must have WINSTA_WRITEATTRIBUTES access to the window station.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.37 CloseClipboard

The CloseClipboard function closes the clipboard.

```
CloseClipboard: procedure;
    @stdcall;
    @returns( "eax" );
    @external( "__imp__CloseClipboard@0" );
```

Parameters

This function has no parameters.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

When the window has finished examining or changing the clipboard, close the clipboard by calling CloseClipboard. This enables other windows to access the clipboard.

Do not place an object on the clipboard after calling CloseClipboard.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.38 CloseDesktop

The CloseDesktop function closes an open handle to a desktop object. A desktop is a secure object contained within a window station object. A desktop has a logical display surface and contains windows, menus and hooks.

```
CloseDesktop: procedure
(
    hDesktop:dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__CloseDesktop@4" );
```

Parameters

hDesktop

[in] Handle to the desktop to close. This can be a handle returned by the CreateDesktop, OpenDesktop, or OpenInputDesktop functions.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

The CloseDesktop function will fail if any thread in the calling process is using the specified desktop handle or if the handle refers to the initial desktop of the calling process.

Requirements

Windows NT/2000: Requires Windows NT 3.51 or later.

Windows 95/98: Unsupported.

3.39 CloseWindow

The CloseWindow function minimizes (but does not destroy) the specified window.

```
CloseWindow: procedure
(
    hWnd            :dword
```

```
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__CloseWindow@4" );
```

Parameters

hWnd

[in] Handle to the window to be minimized.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

The window is minimized by reducing it to the size of an icon and moving the window to the icon area of the screen. The system displays the window's icon instead of the window and draws the window's title below the icon.

To destroy a window, an application must use the DestroyWindow function.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.40 CloseWindowStation

The CloseWindowStation function closes an open window station handle.

```
CloseWindowStation: procedure
(
    hWinSta         :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__CloseWindowStation@4" );
```

Parameters

hWinSta

[in] Handle to the window station to be closed. This handle is returned by the CreateWindowStation and OpenWindowStation functions.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

The CloseWindowStation function will fail if the handle being closed is for the window station assigned to the calling process.

Requirements

Windows NT/2000: Requires Windows NT 3.51 or later.

Windows 95/98: Unsupported.

3.41 CopyAcceleratorTable

The CopyAcceleratorTable function copies the specified accelerator table. This function is used to obtain the accelerator-table data that corresponds to an accelerator-table handle, or to determine the size of the accelerator-table data.

```
CopyAcceleratorTable: procedure
(
        hAccelSrc       :dword;
    var lpAccelDst      :ACCEL;
        cAccelEntries   :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__CopyAcceleratorTableA@12" );
```

Parameters

hAccelSrc

[in] Handle to the accelerator table to copy.

lpAccelDst

[out] Pointer to an array of ACCEL structures that receives the accelerator-table information.

cAccelEntries

[in] Specifies the number of ACCEL structures to copy to the buffer pointed to by the lpAccelDst parameter.

Return Values

If lpAccelDst is NULL, the return value specifies the number of accelerator-table entries in the original table. Otherwise, it specifies the number of accelerator-table entries that were copied.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.42 CopyIcon

The CopyIcon function copies the specified icon from another module to the current module.

```
CopyIcon: procedure
(
    hIcon           :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__CopyIcon@4" );
```

Parameters

hIcon

[in] Handle to the icon to be copied.

Return Values

If the function succeeds, the return value is a handle to the duplicate icon.

If the function fails, the return value is NULL. To get extended error information, call GetLastError.

Remarks

The CopyIcon function enables an application or dynamic-link library (DLL) to get its own handle to an icon owned by another module. If the other module is freed, the application icon will still be able to use the icon.

Before closing, an application must call the DestroyIcon function to free any system resources associated with the icon.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.43 CopyImage

The CopyImage function creates a new image (icon, cursor, or bitmap) and copies the attributes of the specified image to the new one. If necessary, the function stretches the bits to fit the desired size of the new image.

```
CopyImage: procedure
(
    hImage      :dword;
    uType       :dword;
    cxDesired   :dword;
    cyDesired   :dword;
    fuFlags     :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__CopyImage@20" );
```

Parameters

hImage

[in] Handle to the image to be copied.

uType

[in] Specifies the type of image to be copied. This parameter can be one of the following values.

Value	Meaning
IMAGE_BITMAP	Copies a bitmap.
IMAGE_CURSOR	Copies a cursor.
IMAGE_ICON	Copies an icon.

cxDesired

[in] Specifies the desired width, in pixels, of the image. If this is zero, then the returned image will have the same width as the original hImage.

cyDesired

[in] Specifies the desired height, in pixels, of the image. If this is zero, then the returned image will have the same height as the original hImage.

fuFlags

[in] This parameter can be one or more of the following values.

Value	Meaning
LR_COPYDELETEORG	Deletes the original image after creating the copy.
LR_COPYFROMRESOURCE	Tries to reload an icon or cursor resource from the original resource file rather than simply copying the current image. This is useful for creating a different-sized copy when the resource file contains multiple sizes of the resource. Without this flag, stretches the original image to the new size. If this flag is set, uses the size in the resource file closest to the desired size. This will succeed only if hImage was loaded by LoadIcon or LoadCursor, or by LoadImage with the LR_SHARED flag.
LR_COPYRETURNORG	Returns the original hImage if it satisfies the criteria for the copy—that is, correct dimensions and color depth—in which case the LR_COPYDELETEORG flag is ignored. If this flag is not specified, a new object is always created.
LR_CREATEDIBSECTION	If this is set and a new bitmap is created, the bitmap is created as a DIB section. Otherwise, the bitmap image is created as a device-dependent bitmap. This flag is only valid if uType is IMAGE_BITMAP.
LR_MONOCHROME	Creates a new monochrome image.

Return Values

If the function succeeds, the return value is the handle to the newly created image.

If the function fails, the return value is NULL. To get extended error information, call GetLastError.

Remarks

When you are finished using the resource, you can release its associated memory by calling one of the functions in the following table.

Resource	Release function
Bitmap	DeleteObject
Cursor	**DestroyCursor**
Icon	**DestroyIcon**

The system automatically deletes the resource when its process terminates, however, calling the appropriate function saves memory and decreases the size of the process's working set.

Requirements

Windows NT/2000: Requires Windows NT 3.5 or later.

Windows 95/98: Requires Windows 95 or later.

3.44 CopyRect

The CopyRect function copies the coordinates of one rectangle to another.

```
CopyRect: procedure
(
    var lprcDst :RECT;
```

```
    var lprcSrc :RECT
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__CopyRect@8" );
```

Parameters

lprcDst

[out] Pointer to the RECT structure that receives the logical coordinates of the source rectangle.

lprcSrc

[in] Pointer to the RECT structure whose coordinates are to be copied in logical units.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Windows NT/ 2000: To get extended error information, call GetLastError.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.45 CountClipboardFormats

The CountClipboardFormats function retrieves the number of different data formats currently on the clipboard.

```
CountClipboardFormats: procedure;
    @stdcall;
    @returns( "eax" );
    @external( "__imp__CountClipboardFormats@0" );
```

Parameters

This function has no parameters.

Return Values

If the function succeeds, the return value is the number of different data formats currently on the clipboard.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.46 CreateAcceleratorTable

The CreateAcceleratorTable function creates an accelerator table.

```
CreateAcceleratorTable: procedure
(
    var lpaccel     :ACCEL;
        cEntries    :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__CreateAcceleratorTableA@8" );
```

Parameters

lpaccl

[in] Pointer to an array of ACCEL structures that describes the accelerator table.

cEntries

[in] Specifies the number of ACCEL structures in the array.

Return Values

If the function succeeds, the return value is the handle to the created accelerator table; otherwise, it is NULL. To get extended error information, call GetLastError.

Remarks

Before an application closes, it must use the DestroyAcceleratorTable function to destroy each accelerator table that it created by using the CreateAcceleratorTable function.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.47 CreateCaret

The CreateCaret function creates a new shape for the system caret and assigns ownership of the caret to the specified window. The caret shape can be a line, a block, or a bitmap.

```
CreateCaret: procedure
(
    hWnd            :dword;
    hBitmap         :dword;
    nWidth          :dword;
    nHeight         :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__CreateCaret@16" );
```

Parameters

hWnd

[in] Handle to the window that owns the caret.

hBitmap

[in] Handle to the bitmap that defines the caret shape. If this parameter is NULL, the caret is solid. If this param-

eter is (HBITMAP) 1, the caret is gray. If this parameter is a bitmap handle, the caret is the specified bitmap. The bitmap handle must have been created by the CreateBitmap, CreateDIBitmap, or LoadBitmap function.

If hBitmap is a bitmap handle, CreateCaret ignores the nWidth and nHeight parameters; the bitmap defines its own width and height.

nWidth

[in] Specifies the width of the caret in logical units. If this parameter is zero, the width is set to the system-defined window border width. If hBitmap is a bitmap handle, CreateCaret ignores this parameter.

nHeight

[in] Specifies the height, in logical units, of the caret. If this parameter is zero, the height is set to the system-defined window border height. If hBitmap is a bitmap handle, CreateCaret ignores this parameter.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

The nWidth and nHeight parameters specify the caret's width and height, in logical units; the exact width and height, in pixels, depend on the window's mapping mode.

CreateCaret automatically destroys the previous caret shape, if any, regardless of the window that owns the caret. The caret is hidden until the application calls the ShowCaret function to make the caret visible.

The system provides one caret per queue. A window should create a caret only when it has the keyboard focus or is active. The window should destroy the caret before losing the keyboard focus or becoming inactive.

You can retrieve the width or height of the system's window border by using the GetSystemMetrics function, specifying the SM_CXBORDER and SM_CYBORDER values. Using the window border width or height guarantees that the caret will be visible on a high-resolution screen.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.48 CreateCursor

The CreateCursor function creates a cursor having the specified size, bit patterns, and hot spot.

```
CreateCursor: procedure
(
        hInst       :dword;
        xHotSpot    :dword;
        yHostSpot   :dword;
        nWidth      :dword;
        nHeight     :dword;
    var pvANDPlane  :var;
    var pvXORPlane  :var
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__CreateCursor@28" );
```

Parameters

hInst

[in] Handle to the current instance of the application creating the cursor.

xHotSpot

[in] Specifies the horizontal position of the cursor's hot spot.

yHotSpot

[in] Specifies the vertical position of the cursor's hot spot.

nWidth

[in] Specifies the width, in pixels, of the cursor.

nHeight

[in] Specifies the height, in pixels, of the cursor.

pvANDPlane

[in] Pointer to an array of bytes that contains the bit values for the AND mask of the cursor, as in a device-dependent monochrome bitmap.

pvXORPlane

[in] Pointer to an array of bytes that contains the bit values for the XOR mask of the cursor, as in a device-dependent monochrome bitmap.

Return Values

If the function succeeds, the return value is a handle to the cursor.

If the function fails, the return value is NULL. To get extended error information, call GetLastError.

Remarks

The nWidth and nHeight parameters must specify a width and height that are supported by the current display driver, because the system cannot create cursors of other sizes. To determine the width and height supported by the display driver, use the GetSystemMetrics function, specifying the SM_CXCURSOR or SM_CYCURSOR value.

Before closing, an application must call the DestroyCursor function to free any system resources associated with the cursor.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.49 CreateDesktop

The CreateDesktop function creates a new desktop on the window station associated with the calling process. It retrieves a handle that can be used to access the new desktop. The calling process must have an associated window station, either assigned by the system at process creation time or set by SetProcessWindowStation. A desktop is a secure object contained within a window station object. A desktop has a logical display surface and contains windows, menus, and hooks.

```
CreateDesktop: procedure
(
        lpszDesktop     :string;
```

```
        lpszDevice      :string;
    var pDevmode        :DEVMODE;
        dwFlags         :dword;
        dwDesiredAccess :ACCESS_MASK;
    var lpsa            :SECURITY_ATTRIBUTES
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__CreateDesktopA@24" );
```

Parameters

lpszDesktop

[in] Pointer to a null-terminated string specifying the name of the desktop to be created. Desktop names are case-insensitive and may not contain backslash characters (\).

lpszDevice

Reserved; must be NULL.

pDevmode

Reserved; must be NULL.

dwFlags

[in] Specifies how the calling application will cooperate with other applications on the desktop. This parameter can be zero or the following value.

Value	Description
DF_ALLOWOTHERACCOUNTHOOK	Allows processes running in other accounts on the desktop to set hooks in this process.

dwDesiredAccess

[in] Specifies the access rights the returned handle has to the desktop. This parameter must include the DESKTOP_CREATEWINDOW flag because internally CreateDesktop uses the handle to create a window. In addition, you can specify any of the standard access rights, such as READ_CONTROL or WRITE_DAC, and a combination of the following desktop-specific access rights.

Value	Description
DESKTOP_CREATEMENU	Required to create a menu on the desktop.
DESKTOP_CREATEWINDOW	Required to create a window on the desktop.
DESKTOP_ENUMERATE	Required for the desktop to be enumerated.
DESKTOP_HOOKCONTROL	Required to establish any of the window hooks.
DESKTOP_JOURNALPLAYBACK	Required to perform journal playback on the desktop.
DESKTOP_JOURNALRECORD	Required to perform journal recording on the desktop.
DESKTOP_READOBJECTS	Required to read objects on the desktop.
DESKTOP_SWITCHDESKTOP	Required to activate the desktop using the SwitchDesktop function.
DESKTOP_WRITEOBJECTS	Required to write objects on the desktop.

lpsa

[in] Pointer to a SECURITY_ATTRIBUTES structure that determines whether the returned handle can be inherited by child processes. If lpsa is NULL, the handle cannot be inherited.

The lpSecurityDescriptor member of the structure specifies a security descriptor for the new desktop. If lpsa is NULL, the desktop inherits its security descriptor from the parent window station.

Return Values

If the function succeeds, the return value is a handle to the newly created desktop. If the specified desktop already exists, the function succeeds and returns a handle to the existing desktop. When you are finished using the han-

dle, call the CloseDesktop function to close it.

If the function fails, the return value is NULL. To get extended error information, call GetLastError.

Remarks

The CreateDesktop function returns a handle that can be used to access the desktop.

If the dwDesiredAccess parameter specifies the READ_CONTROL, WRITE_DAC, or WRITE_OWNER standard access rights to access the security descriptor of the desktop object, you must also request the DESKTOP_READOBJECTS and DESKTOP_WRITEOBJECTS access rights.

Requirements

Windows NT/2000: Requires Windows NT 3.51 or later.

Windows 95/98: Unsupported.

3.50 CreateDialogIndirectParam

The CreateDialogIndirectParam function creates a modeless dialog box from a dialog box template in memory. Before displaying the dialog box, the function passes an application-defined value to the dialog box procedure as the lParam parameter of the WM_INITDIALOG message. An application can use this value to initialize dialog box controls.

```
CreateDialogIndirectParam: procedure
(
        hInstance       :dword;
    var lpTemplate      :CDLGTEMPLATE;
        hWndParent      :dword;
        lpDialogFunc    :DLGPROC;
        lParamInit      :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__CreateDialogIndirectParamA@20" );
```

Parameters

hInstance

[in] Handle to the module that will create the dialog box.

lpTemplate

[in] Pointer to a global memory object that contains the template CreateDialogIndirectParam uses to create the dialog box. A dialog box template consists of a header that describes the dialog box, followed by one or more additional blocks of data that describe each of the controls in the dialog box. The template can use either the standard format or the extended format.

In a standard template, the header is a DLGTEMPLATE structure followed by additional variable-length arrays. The data for each control consists of a DLGITEMTEMPLATE structure followed by additional variable-length arrays.

In an extended dialog box template, the header uses the DLGTEMPLATEEX format and the control definitions use the DLGITEMTEMPLATEEX format.

After CreateDialogIndirectParam returns, you can free the template, which is only used to get the dialog box started.

hWndParent

[in] Handle to the window that owns the dialog box.

lpDialogFunc

[in] Pointer to the dialog box procedure. For more information about the dialog box procedure, see DialogProc.

lParamInit

[in] Specifies the value to pass to the dialog box in the lParam parameter of the WM_INITDIALOG message.

Return Values

If the function succeeds, the return value is the window handle to the dialog box.

If the function fails, the return value is NULL. To get extended error information, call GetLastError.

Remarks

The CreateDialogIndirectParam function uses the CreateWindowEx function to create the dialog box. CreateDialogIndirectParam then sends a WM_INITDIALOG message to the dialog box procedure. If the template specifies the DS_SETFONT or DS_SHELLFONT style, the function also sends a WM_SETFONT message to the dialog box procedure. The function displays the dialog box if the template specifies the WS_VISIBLE style. Finally, CreateDialogIndirectParam returns the window handle to the dialog box.

After CreateDialogIndirectParam returns, you can use the ShowWindow function to display the dialog box (if it is not already visible). To destroy the dialog box, use the DestroyWindow function. To support keyboard navigation and other dialog box functionality, the message loop for the dialog box must call the IsDialogMessage function.

In a standard dialog box template, the DLGTEMPLATE structure and each of the DLGITEMTEMPLATE structures must be aligned on DWORD boundaries. The creation data array that follows a DLGITEMTEMPLATE structure must also be aligned on a DWORD boundary. All of the other variable-length arrays in the template must be aligned on WORD boundaries.

In an extended dialog box template, the DLGTEMPLATEEX header and each of the DLGITEMTEMPLATEEX control definitions must be aligned on DWORD boundaries. The creation data array, if any, that follows a DLGITEMTEMPLATEEX structure must also be aligned on a DWORD boundary. All of the other variable-length arrays in the template must be aligned on WORD boundaries.

All character strings in the dialog box template, such as titles for the dialog box and buttons, must be Unicode strings. To create code that works on both Windows 95/98 and Windows NT/Windows 2000, use the MultiByteToWideChar function to generate these Unicode strings.

Windows 95/98: The system can support a maximum of 255 controls per dialog box template. To place more than 255 controls in a dialog box, create the controls in the WM_INITDIALOG message handler rather than placing them in the template.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.51 CreateDialogParam

The CreateDialogParam function creates a modeless dialog box from a dialog box template resource. Before displaying the dialog box, the function passes an application-defined value to the dialog box procedure as the lParam parameter of the WM_INITDIALOG message. An application can use this value to initialize dialog box controls.

```
CreateDialogParam: procedure
(
    hInstance           :dword;
    lpTemplateName      :string;
    hWndParent          :dword;
    lpDialogFunc        :DLGPROC;
    dwInitParam         :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__CreateDialogParamA@20" );
```

Parameters

hInstance

[in] Handle to the module whose executable file contains the dialog box template.

lpTemplateName

[in] Specifies the dialog box template. This parameter is either the pointer to a null-terminated character string that specifies the name of the dialog box template or an integer value that specifies the resource identifier of the dialog box template. If the parameter specifies a resource identifier, its high-order word must be zero and low-order word must contain the identifier. You can use the MAKEINTRESOURCE macro to create this value.

hWndParent

[in] Handle to the window that owns the dialog box.

lpDialogFunc

[in] Pointer to the dialog box procedure. For more information about the dialog box procedure, see DialogProc.

dwInitParam

[in] Specifies the value to pass to the dialog box procedure in the lParam parameter in the WM_INITDIALOG message.

Return Values

If the function succeeds, the return value is the window handle to the dialog box.

If the function fails, the return value is NULL. To get extended error information, call GetLastError.

Remarks

The CreateDialogParam function uses the CreateWindowEx function to create the dialog box. CreateDialog-Param then sends a WM_INITDIALOG message (and a WM_SETFONT message if the template specifies the DS_SETFONT or DS_SHELLFONT style) to the dialog box procedure. The function displays the dialog box if the template specifies the WS_VISIBLE style. Finally, CreateDialogParam returns the window handle of the dialog box.

After CreateDialogParam returns, the application displays the dialog box (if it is not already displayed) using the ShowWindow function. The application destroys the dialog box by using the DestroyWindow function. To support keyboard navigation and other dialog box functionality, the message loop for the dialog box must call the IsDialogMessage function.

Windows 95/98: The system can support a maximum of 255 controls per dialog box template. To place more than 255 controls in a dialog box, create the controls in the WM_INITDIALOG message handler rather than placing them in the template.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.52 CreateIcon

The CreateIcon function creates an icon that has the specified size, colors, and bit patterns.

```
CreateIcon: procedure
(
        hInstance       :dword;
        nWidth          :dword;
        nHeight         :dword;
        cPlanes         :dword;
        cBitsPixel      :dword;
    var lpbANDbits      :var;
    var lpbXORbits      :var
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__CreateIcon@28" );
```

Parameters

hInstance

[in] Handle to the instance of the module creating the icon.

nWidth

[in] Specifies the width, in pixels, of the icon.

nHeight

[in] Specifies the height, in pixels, of the icon.

cPlanes

[in] Specifies the number of planes in the XOR bitmask of the icon.

cBitsPixel

[in] Specifies the number of bits-per-pixel in the XOR bitmask of the icon.

lpbANDbits

[in] Pointer to an array of bytes that contains the bit values for the AND bitmask of the icon. This bitmask describes a monochrome bitmap.

lpbXORbits

[in] Pointer to an array of bytes that contains the bit values for the XOR bitmask of the icon. This bitmask describes a monochrome or device-dependent color bitmap.

Return Values

If the function succeeds, the return value is a handle to an icon.

If the function fails, the return value is NULL. To get extended error information, call GetLastError.

Remarks

The nWidth and nHeight parameters must specify a width and height supported by the current display driver, because the system cannot create icons of other sizes. To determine the width and height supported by the display driver, use the GetSystemMetrics function, specifying the SM_CXICON or SM_CYICON value.

CreateIcon applies the following truth table to the AND and XOR bitmasks:

AND bitmask	XOR bitmask	Display
0	0	Black
0	1	White
1	0	Screen
1	1	Reverse screen

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.53 CreateIconFromResource

The CreateIconFromResource function creates an icon or cursor from resource bits describing the icon.

To specify a desired height or width, use the CreateIconFromResourceEx function.

```
CreateIconFromResource: procedure
(
    var presbits    :var;
        dwResSize   :dword;
        fIcon       :boolean;
        dwVer       :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__CreateIconFromResource@16" );
```

Parameters

presbits

[in] Pointer to a buffer containing the icon or cursor resource bits. These bits are typically loaded by calls to the LookupIconIdFromDirectory (in Windows 95 you can also call LookupIconIdFromDirectoryEx) and LoadResource functions.

dwResSize

[in] Specifies the size, in bytes, of the set of bits pointed to by the presbits parameter.

fIcon

[in] Specifies whether an icon or a cursor is to be created. If this parameter is TRUE, an icon is to be created. If it is FALSE, a cursor is to be created.

dwVer

[in] Specifies the version number of the icon or cursor format for the resource bits pointed to by the presbits parameter. This parameter can be one of the following values:

Format	dwVer
Windows 2.*x*	0x00020000
Windows 3.x	0x00030000

All Win32-based applications use the Windows 3.x format for icons and cursors.

Return Values

If the function succeeds, the return value is a handle to the icon or cursor.

If the function fails, the return value is NULL. To get extended error information, call GetLastError.

Remarks

The CreateIconFromResource, CreateIconFromResourceEx, CreateIconIndirect, GetIconInfo, LookupIconId-FromDirectory, and LookupIconIdFromDirectoryEx functions allow shell applications and icon browsers to examine and use resources throughout the system.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.54 CreateIconFromResourceEx

The CreateIconFromResourceEx function creates an icon or cursor from resource bits describing the icon.

```
CreateIconFromResourceEx: procedure
(
    var pbIconBits  :var;
        cbIconBits  :dword;
        fIcon       :boolean;
        dwVersion   :dword;
        cxDesired   :dword;
        cyDesired   :dword;
        uFlags      :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__CreateIconFromResourceEx@28" );
```

Parameters

pbIconBits

[in] Pointer to a buffer containing the icon or cursor resource bits. These bits are typically loaded by calls to the LookupIconIdFromDirectoryEx and LoadResource functions.

cbIconBits

[in] Specifies the size, in bytes, of the set of bits pointed to by the pbIconBits parameter.

fIcon

[in] Specifies whether an icon or a cursor is to be created. If this parameter is TRUE, an icon is to be created. If it is FALSE, a cursor is to be created.

dwVersion

[in] Specifies the version number of the icon or cursor format for the resource bits pointed to by the pbIcon-Bits parameter. This parameter can be one of the following values:

Format	dwVersion
Windows 2.*x*	0x00020000
Windows 3.x	0x00030000

All Win32-based applications use the Windows 3.x format for icons and cursors.

cxDesired

[in] Specifies the desired width, in pixels, of the icon or cursor. If this parameter is zero, the function uses the SM_CXICON or SM_CXCURSOR system metric value to set the width.

cyDesired

[in] Specifies the desired height, in pixels, of the icon or cursor. If this parameter is zero, the function uses the SM_CYICON or SM_CYCURSOR system metric value to set the height.

uFlags

[in] Specifies a combination of the following values:

Value	Meaning
LR_DEFAULTCOLOR	Uses the default color format.
LR_MONOCHROME	Creates a monochrome icon or cursor.

Return Values

If the function succeeds, the return value is a handle to the icon or cursor.

If the function fails, the return value is NULL. To get extended error information, call GetLastError.

Remarks

The CreateIconFromResource, CreateIconFromResourceEx, CreateIconIndirect, GetIconInfo, and LookupIconIdFromDirectoryEx functions allow shell applications and icon browsers to examine and use resources throughout the system.

Requirements

Windows NT/2000: Requires Windows NT 4.0 or later.

Windows 95/98: Requires Windows 95 or later.

3.55 CreateIconIndirect

The CreateIconIndirect function creates an icon or cursor from an ICONINFO structure.

```
CreateIconIndirect: procedure
(
    var piconinfo   :var
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__CreateIconIndirect@4" );
```

Parameters

piconinfo

[in] Pointer to an ICONINFO structure the function uses to create the icon or cursor.

Return Values

If the function succeeds, the return value is a handle to the icon or cursor that is created.

If the function fails, the return value is NULL. To get extended error information, call GetLastError.

Remarks

The system copies the bitmaps in the ICONINFO structure before creating the icon or cursor. Because the system may temporarily select the bitmaps in a device context, the hbmMask and hbmColor members of the ICONINFO structure should not already be selected into a device context. The application must continue to manage the original bitmaps and delete them when they are no longer necessary.

When you are finished using the icon, destroy it using the DestroyIcon function.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.56 CreateMDIWindow

The CreateMDIWindow function creates a multiple document interface (MDI) child window.

```
CreateMDIWindow: procedure
(
    lpClassName     :string;
    lpWindowName    :string;
    dwStyle         :dword;
    X               :dword;
    Y               :dword;
    nWidth          :dword;
    nHeight         :dword;
    hWndParent      :dword;
    hInstance       :dword;
    _lParam         :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__CreateMDIWindowA@40" );
```

Parameters

lpClassName

[in] Pointer to a null-terminated string specifying the window class of the MDI child window. The class name must have been registered by a call to the RegisterClassEx function.

lpWindowName

[in] Pointer to a null-terminated string that represents the window name. The system displays the name in the title bar of the child window.

dwStyle

[in] Specifies the style of the MDI child window. If the MDI client window is created with the MDIS_ALLCHILDSTYLES window style, this parameter can be any combination of the window styles listed in the description of the CreateWindow function. Otherwise, this parameter can be one or more of the following values.

Value	Meaning
WS_MINIMIZE	Creates an MDI child window that is initially minimized.
WS_MAXIMIZE	Creates an MDI child window that is initially maximized.
WS_HSCROLL	Creates an MDI child window that has a horizontal scroll bar.
WS_VSCROLL	Creates an MDI child window that has a vertical scroll bar.

X

[in] Specifies the initial horizontal position, in client coordinates, of the MDI child window. If this parameter is CW_USEDEFAULT, the MDI child window is assigned the default horizontal position.

Y

[in] Specifies the initial vertical position, in client coordinates, of the MDI child window. If this parameter is

CW_USEDEFAULT, the MDI child window is assigned the default vertical position.

nWidth

[in] Specifies the initial width, in device units, of the MDI child window. If this parameter is CW_USEDEFAULT, the MDI child window is assigned the default width.

nHeight

[in] Specifies the initial height, in device units, of the MDI child window. If this parameter is set to CW_USEDEFAULT, the MDI child window is assigned the default height.

hWndParent

[in] Handle to the MDI client window that will be the parent of the new MDI child window.

hInstance

[in] Handle to the instance of the application creating the MDI child window.

lParam

[in] Specifies an application-defined value.

Return Values

If the function succeeds, the return value is the handle to the created window.

If the function fails, the return value is NULL. To get extended error information, call GetLastError.

Remarks

Using the CreateMDIWindow function is similar to sending the WM_MDICREATE message to an MDI client window, except that the function can create an MDI child window in a different thread, while the message cannot.

Windows 95: The system can support a maximum of 16,364 window handles.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.57 CreateMenu

The CreateMenu function creates a menu. The menu is initially empty, but it can be filled with menu items by using the InsertMenuItem, AppendMenu, and InsertMenu functions.

```
CreateMenu: procedure;
    @stdcall;
    @returns( "eax" );
    @external( "__imp__CreateMenu@0" );
```

Parameters

This function has no parameters.

Return Values

If the function succeeds, the return value is a handle to the newly created menu.

If the function fails, the return value is NULL. To get extended error information, call GetLastError.

Remarks

Resources associated with a menu that is assigned to a window are freed automatically. If the menu is not assigned to a window, an application must free system resources associated with the menu before closing. An application frees menu resources by calling the DestroyMenu function.

Windows 95: The system can support a maximum of 16,364 menu handles.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.58 CreatePopupMenu

The CreatePopupMenu function creates a drop-down menu, submenu, or shortcut menu. The menu is initially empty. You can insert or append menu items by using the InsertMenuItem function. You can also use the Insert-Menu function to insert menu items and the AppendMenu function to append menu items.

```
CreatePopupMenu: procedure;
    @stdcall;
    @returns( "eax" );
    @external( "__imp__CreatePopupMenu@0" );
```

Parameters

This function has no parameters.

Return Values

If the function succeeds, the return value is a handle to the newly created menu.

If the function fails, the return value is NULL. To get extended error information, call GetLastError.

Remarks

The application can add the new menu to an existing menu, or it can display a shortcut menu by calling the TrackPopupMenuEx or TrackPopupMenu functions.

Resources associated with a menu that is assigned to a window are freed automatically. If the menu is not assigned to a window, an application must free system resources associated with the menu before closing. An application frees menu resources by calling the DestroyMenu function.

Windows 95: The system can support a maximum of 16,364 menu handles.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.59 CreateWindowEx

The CreateWindowEx function creates an overlapped, pop-up, or child window with an extended window style; otherwise, this function is identical to the CreateWindow function. For more information about creating a window and for full descriptions of the other parameters of CreateWindowEx, see CreateWindow.

```
CreateWindowEx: procedure
(
        dwExStyle       :dword;
        lpClassName     :string;
        lpWindowName    :string;
        dwStyle         :dword;
        x               :dword;
        y               :dword;
        nWidth          :dword;
        nHeight         :dword;
        hWndParent      :dword;
        hMenu           :dword;
        hInstance       :dword;
    var lpParam         :var
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__CreateWindowExA@48" );
```

Parameters

dwExStyle

[in] Specifies the extended window style of the window being created. This parameter can be one or more of the following values.

Style	Meaning
WS_EX_ACCEPTFILES	Specifies that a window created with this style accepts drag-drop files.
WS_EX_APPWINDOW	Forces a top-level window onto the taskbar when the window is visible.
WS_EX_CLIENTEDGE	Specifies that a window has a border with a sunken edge.
WS_EX_COMPOSITED	Whistler: Paints all descendants of a window in bottom-to-top painting order using double-buffering. For more information, see Remarks.
WS_EX_CONTEXTHELP	Includes a question mark in the title bar of the window. When the user clicks the question mark, the cursor changes to a question mark with a pointer. If the user then clicks a child window, the child receives a WM_HELP message. The child window should pass the message to the parent window procedure, which should call the WinHelp function using the HELP_WM_HELP command. The Help application displays a pop-up window that typically contains help for the child window. WS_EX_CONTEXTHELP cannot be used with the WS_MAXIMIZEBOX or WS_MINIMIZEBOX styles.
WS_EX_CONTROLPARENT	The window itself contains child windows that should take part in dialog box navigation. If this style is specified, the dialog manager recurses into children of this window when performing navigation operations such as handling the TAB key, an arrow key, or a keyboard mnemonic.
WS_EX_DLGMODALFRAME	Creates a window that has a double border; the window can, optionally, be created with a title bar by specifying the WS_CAPTION style in the dwStyle parameter.
WS_EX_LAYERED	Windows 2000: Creates a layered window. Note that this cannot be used for child windows.

WS_EX_LAYOUTRTL	Windows 2000 and the Arabic and Hebrew versions of Windows 98 and Windows Me: Creates a window whose horizontal origin is on the right edge. Increasing horizontal values advance to the left.
WS_EX_LEFT	Creates a window that has generic left-aligned properties. This is the default.
WS_EX_LEFTSCROLLBAR	If the shell language is Hebrew, Arabic, or another language that supports reading order alignment, the vertical scroll bar (if present) is to the left of the client area. For other languages, the style is ignored.
WS_EX_LTRREADING	The window text is displayed using left-to-right reading-order properties. This is the default.
WS_EX_MDICHILD	Creates an MDI child window.
WS_EX_NOACTIVATE	Windows 2000: A top-level window created with this style does not become the foreground window when the user clicks it. The system does not bring this window to the foreground when the user minimizes or closes the foreground window. To activate the window, use the SetActiveWindow or SetForegroundWindow function. The window does not appear on the taskbar by default. To force the window to appear on the taskbar, use the WS_EX_APPWINDOW style.
WS_EX_NOINHERITLAYOUT	Windows 2000: A window created with this style does not pass its window layout to its child windows.
WS_EX_NOPARENTNOTIFY	Specifies that a child window created with this style does not send the WM_PARENTNOTIFY message to its parent window when it is created or destroyed.
WS_EX_OVERLAPPEDWINDOW	Combines the WS_EX_CLIENTEDGE and WS_EX_WINDOWEDGE styles.
WS_EX_PALETTEWINDOW	Combines the WS_EX_WINDOWEDGE, WS_EX_TOOLWINDOW, and WS_EX_TOPMOST styles.
WS_EX_RIGHT	The window has generic "right-aligned" properties. This depends on the window class. This style has an effect only if the shell language is Hebrew, Arabic, or another language that supports reading-order alignment; otherwise, the style is ignored.
WS_EX_RIGHTSCROLLBAR	Vertical scroll bar (if present) is to the right of the client area. This is the default.
WS_EX_RTLREADING	If the shell language is Hebrew, Arabic, or another language that supports reading-order alignment, the window text is displayed using right-to-left reading-order properties. For other languages, the style is ignored.
WS_EX_STATICEDGE	Creates a window with a three-dimensional border style intended to be used for items that do not accept user input.
WS_EX_TOOLWINDOW	Creates a tool window; that is, a window intended to be used as a floating toolbar. A tool window has a title bar that is shorter than a normal title bar, and the window title is drawn using a smaller font. A tool window does not appear in the taskbar or in the dialog that appears when the user presses ALT+TAB. If a tool window has a system menu, its icon is not displayed on the title bar. However, you can display the system menu by right-clicking or by typing ALT+SPACE.

WS_EX_TOPMOST
: Specifies that a window created with this style should be placed above all non-topmost windows and should stay above them, even when the window is deactivated. To add or remove this style, use the SetWindowPos function.

WS_EX_TRANSPARENT
: Specifies that a window created with this style should not be painted until siblings beneath the window (that were created by the same thread) have been painted. The window appears transparent because the bits of underlying sibling windows have already been painted.

  To achieve transparency without these restrictions, use the SetWindowRgn function.

WS_EX_WINDOWEDGE
: Specifies that a window has a border with a raised edge.

Using the WS_EX_RIGHT style for static or edit controls has the same effect as using the SS_RIGHT or ES_RIGHT style, respectively. Using this style with button controls has the same effect as using BS_RIGHT and BS_RIGHTBUTTON styles.

lpClassName

[in] Pointer to a null-terminated string or a class atom created by a previous call to the RegisterClass or RegisterClassEx function. The atom must be in the low-order word of lpClassName; the high-order word must be zero.

If lpClassName is a string, it specifies the window class name. The class name can be any name registered with RegisterClass or RegisterClassEx, provided that the module that registers the class is also the module that creates the window. The class name can also be any of the predefined system class names.

lpWindowName

[in] Pointer to a null-terminated string that specifies the window name.

If the window style specifies a title bar, the window title pointed to by lpWindowName is displayed in the title bar. When using CreateWindow to create controls, such as buttons, check boxes, and static controls, use lpWindowName to specify the text of the control. When creating a static control with the SS_ICON style, use lpWindowName to specify the icon name or identifier. To specify an identifier, use the syntax "#num".

dwStyle

[in] Specifies the style of the window being created. This parameter can be a combination of window styles, plus the control styles indicated in the Remarks section.

x

[in] Specifies the initial horizontal position of the window. For an overlapped or pop-up window, the x parameter is the initial x-coordinate of the window's upper-left corner, in screen coordinates. For a child window, x is the x-coordinate of the upper-left corner of the window relative to the upper-left corner of the parent window's client area.

If x is set to CW_USEDEFAULT, the system selects the default position for the window's upper-left corner and ignores the y parameter. CW_USEDEFAULT is valid only for overlapped windows; if it is specified for a pop-up or child window, the x and y parameters are set to zero.

y

[in] Specifies the initial vertical position of the window. For an overlapped or pop-up window, the y parameter is the initial y-coordinate of the window's upper-left corner, in screen coordinates. For a child window, y is the initial y-coordinate of the upper-left corner of the child window relative to the upper-left corner of the parent window's client area. For a list box, y is the initial y-coordinate of the upper-left corner of the list box's client area relative to the upper-left corner of the parent window's client area.

If an overlapped window is created with the WS_VISIBLE style bit set and the x parameter is set to CW_USEDEFAULT, the system ignores the y parameter.

nWidth

[in] Specifies the width, in device units, of the window. For overlapped windows, nWidth is the window's width, in screen coordinates, or CW_USEDEFAULT. If nWidth is CW_USEDEFAULT, the system selects a default width and height for the window; the default width extends from the initial x-coordinates to the right edge of the screen; the default height extends from the initial y-coordinate to the top of the icon area. CW_USEDEFAULT is valid only for overlapped windows; if CW_USEDEFAULT is specified for a pop-up or child window, the nWidth and nHeight parameter are set to zero.

nHeight

[in] Specifies the height, in device units, of the window. For overlapped windows, nHeight is the window's height, in screen coordinates. If the nWidth parameter is set to CW_USEDEFAULT, the system ignores nHeight.

hWndParent

[in] Handle to the parent or owner window of the window being created. To create a child window or an owned window, supply a valid window handle. This parameter is optional for pop-up windows.

Windows 2000: To create a message-only window, supply HWND_MESSAGE or a handle to an existing message-only window.

hMenu

[in] Handle to a menu, or specifies a child-window identifier, depending on the window style. For an overlapped or pop-up window, hMenu identifies the menu to be used with the window; it can be NULL if the class menu is to be used. For a child window, hMenu specifies the child-window identifier, an integer value used by a dialog box control to notify its parent about events. The application determines the child-window identifier; it must be unique for all child windows with the same parent window.

hInstance

Windows 95/98: [in] Handle to the instance of the module to be associated with the window.

Windows NT/2000: This value is ignored.

lpParam

[in] Pointer to a value to be passed to the window through the CREATESTRUCT structure passed in the lParam parameter the WM_CREATE message. If an application calls CreateWindow to create a multiple document interface (MDI) client window, lpParam must point to a CLIENTCREATESTRUCT structure.

Return Values

If the function succeeds, the return value is a handle to the new window.

If the function fails, the return value is NULL. To get extended error information, call GetLastError.

This function typically fails for one of the following reasons:

- an invalid parameter value
- the system class was registered by a different module
- the WH_CBT hook is installed and returns a failure code
- the window procedure fails for WM_CREATE or WM_NCCREATE

Remarks

The CreateWindowEx function sends WM_NCCREATE, WM_NCCALCSIZE, and WM_CREATE messages to the window being created.

For information on controlling whether the Taskbar displays a button for the created window, see Visibility of Taskbar Buttons .

The following predefined control classes can be specified in the lpClassName parameter. Note the corresponding

control styles you can use in the dwStyle parameter.

Class	Meaning
BUTTON	Designates a small rectangular child window that represents a button the user can click to turn it on or off. Button controls can be used alone or in groups, and they can either be labeled or appear without text. Button controls typically change appearance when the user clicks them. For more information, see Buttons . For a table of the button styles you can specify in the dwStyle parameter, see Button Styles.
COMBOBOX	Designates a control consisting of a list box and a selection field similar to an edit control. When using this style, an application should either display the list box at all times or enable a drop-down list box. If the list box is visible, typing characters into the selection field highlights the first list box entry that matches the characters typed. Conversely, selecting an item in the list box displays the selected text in the selection field. For more information, see Combo Boxes. For a table of the combo box styles you can specify in the dwStyle parameter, see Combo Box Styles .
EDIT	Designates a rectangular child window into which the user can type text from the keyboard. The user selects the control and gives it the keyboard focus by clicking it or moving to it by pressing the TAB key. The user can type text when the edit control displays a flashing caret; use the mouse to move the cursor, select characters to be replaced, or position the cursor for inserting characters; or use the BACKSPACE key to delete characters. For more information, see Edit Controls. For a table of the edit control styles you can specify in the dwStyle parameter, see Edit Control Styles.
LISTBOX	Designates a list of character strings. Specify this control whenever an application must present a list of names, such as filenames, from which the user can choose. The user can select a string by clicking it. A selected string is highlighted, and a notification message is passed to the parent window. For more information, see List Boxes. For a table of the list box styles you can specify in the dwStyle parameter, see List Box Styles.
MDICLIENT	Designates an MDI client window. This window receives messages that control the MDI application's child windows. The recommended style bits are WS_CLIPCHILDREN and WS_CHILD. Specify the WS_HSCROLL and WS_VSCROLL styles to create an MDI client window that allows the user to scroll MDI child windows into view. For more information, see Multiple Document Interface.

RichEdit	Designates a Rich Edit version 1.0 control. This window lets the user view and edit text with character and paragraph formatting, and can include embedded COM objects. For more information, see Rich Edit Controls. For a table of the rich edit control styles you can specify in the dwStyle parameter, see Rich Edit Control Styles.
RICHEDIT_CLASS	Designates a Rich Edit version 2.0 control. This controls let the user view and edit text with character and paragraph formatting, and can include embedded COM objects. For more information, see Rich Edit Controls. For a table of the rich edit control styles you can specify in the dwStyle parameter, see Rich Edit Control Styles.
SCROLLBAR	Designates a rectangle that contains a scroll box and has direction arrows at both ends. The scroll bar sends a notification message to its parent window whenever the user clicks the control. The parent window is responsible for updating the position of the scroll box, if necessary. For more information, see Scroll Bars. For a table of the scroll bar control styles you can specify in the dwStyle parameter, see Scroll Bar Control Styles.
STATIC	Designates a simple text field, box, or rectangle used to label, box, or separate other controls. Static controls take no input and provide no output. For more information, see Static Controls. For a table of the static control styles you can specify in the dwStyle parameter, see Static Control Styles.

Windows 95: The system can support a maximum of 16,364 window handles.

Windows 2000: The WS_EX_NOACTIVATE value for dwExStyle prevents foreground activation by the system. To prevent queue activation when the user clicks on the window, you must process the WM_MOUSEACTIVATE message appropriately. To bring the window to the foreground or to activate it programmatically, use SetForegroundWindow or SetActiveWindow. Returning FALSE to WM_NCACTIVATE prevents the window from losing queue activation. However, the return value is ignored at activation time.

Whistler: With WS_EX_COMPOSITED set, all descendants of a window get bottom-to-top painting order using double-buffering. Bottom-to-top painting order allows a descendent window to have translucency (alpha) and transparency (color-key) effects, but only if the descendent window also has the WS_EX_TRANSPARENT bit set. Double-buffering allows the window and its descendents to be painted without flicker.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.60 CreateWindowStation

The CreateWindowStation function creates a window station object. It retrieves a handle that can be used to access the window station. A window station is a secure object that contains a set of global atoms, a clipboard, and a set of desktop objects.

```
CreateWindowStation: procedure
```

```
(
        lpwinsta        :string;
        dwReserved      :dword;
        dwDesiredAccess :ACCESS_MASK;
    var lpsa            :SECURITY_ATTRIBUTES
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__CreateWindowStationA@16" );
```

Parameters

lpwinsta

[in] Pointer to a null-terminated string specifying the name of the window station to be created. Window station names are case-insensitive and cannot contain backslash characters (\). Only members of the Administrators group are allowed to specify a name. If lpwinsta is NULL, the system forms a window station name using the logon session identifier for the calling process. To get this name, call the GetUserObjectInformation function.

dwReserved

Reserved; must be zero.

dwDesiredAccess

[in] Specifies the type of access to the window station. This parameter can be one or more of the following values.

Value	Description
WINSTA_ACCESSCLIPBOARD	Required to use the clipboard.
WINSTA_ACCESSGLOBALATOMS	Required to manipulate global atoms.
WINSTA_CREATEDESKTOP	Required to create new desktop objects on the window station.
WINSTA_ENUMDESKTOPS	Required to enumerate existing desktop objects.
WINSTA_ENUMERATE	Required for the window station to be enumerated.
WINSTA_EXITWINDOWS	Required to successfully call the ExitWindows or ExitWindowsEx functions.
WINSTA_READATTRIBUTES	Required to read the attributes of a window station object.
WINSTA_READSCREEN	Required to access screen contents.
WINSTA_WRITEATTRIBUTES	Required to modify the attributes of a window station object.

lpsa

[in] Pointer to a SECURITY_ATTRIBUTES structure that determines whether the returned handle can be inherited by child processes. If lpsa is NULL, the handle cannot be inherited.

The lpSecurityDescriptor member of the structure specifies a security descriptor for the new window station. If lpsa is NULL, the window station (and any desktops created within the window) gets a security descriptor that grants GENERIC_ALL access to all users.

Return Values

If the function succeeds, the return value is a handle to the newly created window station. If the specified window station already exists, the function succeeds and returns a handle to the existing window station.

If the function fails, the return value is NULL. To get extended error information, call GetLastError.

Requirements

Windows NT/2000: Requires Windows NT 3.51 or later.

Windows 95/98: Unsupported.

3.61 DdeAbandonTransaction

The DdeAbandonTransaction function abandons the specified asynchronous transaction and releases all resources associated with the transaction.

```
DdeAbandonTransaction: procedure
(
    idInst          :dword;
    hConv           :dword;
    idTransactions  :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__DdeAbandonTransaction@12" );
```

Parameters

idInst

[in] Specifies the application instance identifier obtained by a previous call to the DdeInitialize function.

hConv

[in] Handle to the conversation in which the transaction was initiated. If this parameter is 0L, all transactions are abandoned (that is, the idTransaction parameter is ignored).

idTransaction

[in] Specifies the identifier of the transaction to abandon. If this parameter is 0L, all active transactions in the specified conversation are abandoned.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Errors

The DdeGetLastError function can be used to get the error code, which can be one of the following values:

DMLERR_DLL_NOT_INITIALIZED

DMLERR_INVALIDPARAMETER

DMLERR_NO_ERROR

DMLERR_UNFOUND_QUEUE_ID

Remarks

Only a dynamic data exchange (DDE) client application should call DdeAbandonTransaction. If the server application responds to the transaction after the client has called DdeAbandonTransaction, the system discards the transaction results. This function has no effect on synchronous transactions.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.62 DdeAccessData

The DdeAccessData function provides access to the data in the specified dynamic data exchange (DDE) object. An application must call the DdeUnaccessData function when it has finished accessing the data in the object.

```
DdeAccessData: procedure
(
        hData       :dword;
    var pcbDataSize :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__DdeAccessData@8" );
```

Parameters

hData

[in] Handle to the DDE object to access.

pcbDataSize

[out] Pointer to a variable that receives the size, in bytes, of the DDE object identified by the hData parameter. If this parameter is NULL, no size information is returned.

Return Values

If the function succeeds, the return value is a pointer to the first byte of data in the DDE object.

If the function fails, the return value is NULL.

Errors

The DdeGetLastError function can be used to get the error code, which can be one of the following values:

DMLERR_DLL_NOT_INITIALIZED

DMLERR_INVALIDPARAMETER

DMLERR_NO_ERROR

Remarks

If the hData parameter has not been passed to a Dynamic Data Exchange Management Library (DDEML) function, an application can use the pointer returned by DdeAccessData for read-write access to the DDE object. If hData has already been passed to a DDEML function, the pointer should be used only for read access to the memory object.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.63 DdeAddData

The DdeAddData function adds data to the specified dynamic data exchange (DDE) object. An application can add data starting at any offset from the beginning of the object. If new data overlaps data already in the object, the new data overwrites the old data in the bytes where the overlap occurs. The contents of locations in the object that have not been written to are undefined.

```
DdeAddData: procedure
(
        hData       :dword;
    var pSrc        :var;
        cb          :dword;
        cbOff       :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__DdeAddData@16" );
```

Parameters

hData

[in] Handle to the DDE object that receives additional data.

pSrc

[in] Pointer to a buffer containing the data to add to the DDE object.

cb

[in] Specifies the length, in bytes, of the data to be added to the DDE object, including the terminating NULL, if the data is a string.

cbOff

[in] Specifies an offset, in bytes, from the beginning of the DDE object. The additional data is copied to the object beginning at this offset.

Return Values

If the function succeeds, the return value is a new handle to the DDE object. The new handle is used in all references to the object.

If the function fails, the return value is zero.

Errors

The DdeGetLastError function can be used to get the error code, which can be one of the following values:

DMLERR_DLL_NOT_INITIALIZED

DMLERR_INVALIDPARAMETER

DMLERR_MEMORY_ERROR

DMLERR_NO_ERROR

Remarks

After a data handle has been used as a parameter in another Dynamic Data Exchange Management Library function or has been returned by a DDE callback function, the handle can be used only for read access to the DDE object identified by the handle.

If the amount of memory originally allocated is less than is needed to hold the added data, DdeAddData reallocates a global memory object of the appropriate size.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.64 DdeClientTransaction

The DdeClientTransaction function begins a data transaction between a client and a server. Only a dynamic data exchange (DDE) client application can call this function, and the application can use it only after establishing a conversation with the server.

```
DdeClientTransaction: procedure
(
    var pData       :var;
        cbData      :dword;
        hConv       :dword;
        hszItem     :dword;
        wFmt        :dword;
        wType       :dword;
        dwTimeOut   :dword;
    var pdwResult   :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__DdeClientTransaction@32" );
```

Parameters

pData

[in] Pointer to the beginning of the data the client must pass to the server.

Optionally, an application can specify the data handle (HDDEDATA) to pass to the server and in that case the cbData parameter should be set to -1. This parameter is required only if the wType parameter is XTYP_EXECUTE or XTYP_POKE. Otherwise, this parameter should be NULL.

For the optional usage of this parameter, XTYP_POKE transactions where pData is a data handle, the handle must have been created by a previous call to the DdeCreateDataHandle function, employing the same data format specified in the wFmt parameter.

cbData

[in] Specifies the length, in bytes, of the data pointed to by the pData parameter, including the terminating NULL, if the data is a string. A value of -1 indicates that pData is a data handle that identifies the data being sent.

hConv

[in] Handle to the conversation in which the transaction is to take place.

hszItem

[in] Handle to the data item for which data is being exchanged during the transaction. This handle must have been created by a previous call to the DdeCreateStringHandle function. This parameter is ignored (and should be set to 0L) if the wType parameter is XTYP_EXECUTE.

wFmt

[in] Specifies the standard clipboard format in which the data item is being submitted or requested.

If the transaction specified by the wType parameter does not pass data or is XTYP_EXECUTE, this parameter should be zero.

If the transaction specified by the wType parameter references non-execute DDE data (XTYP_POKE, XTYP_ADVSTART, XTYP_ADVSTOP, XTYP_REQUEST), the wFmt value must be either a valid predefined (CF_) DDE format or a valid registered clipboard format.

wType

[in] Specifies the transaction type. This parameter can be one of the following values.

Value	Meaning
XTYP_ADVSTART	Begins an advise loop. Any number of distinct advise loops can exist within a conversation. An application can alter the advise loop type by combining the **XTYP_ADVSTART transaction type with one or more of the following flags:**

Flag	Meaning
XTYPF_NODATA	Instructs the server to notify the client of any data changes without actually sending the data. This flag gives the client the option of ignoring the notification or requesting the changed data from the server.
XTYPF_ACKREQ	Instructs the server to wait until the client acknowledges that it received the previous data item before sending the next data item. This flag prevents a fast server from sending data faster than the client can process it.

XTYP_ADVSTOP	Ends an advise loop.
XTYP_EXECUTE	Begins an execute transaction.
XTYP_POKE	Begins a poke transaction.
XTYP_REQUEST	Begins a request transaction.

dwTimeout

[in] Specifies the maximum length of time, in milliseconds, that the client will wait for a response from the server application in a synchronous transaction. This parameter should be TIMEOUT_ASYNC for asynchronous transactions.

pdwResult

[out] Pointer to a variable that receives the result of the transaction. An application that does not check the result can use NULL for this value. For synchronous transactions, the low-order word of this variable contains any applicable DDE_ flags resulting from the transaction. This provides support for applications dependent on DDE_APPSTATUS bits. It is, however, recommended that applications no longer use these bits because they may not be supported in future versions of the Dynamic Data Exchange Management Library (DDEML). For asynchronous transactions, this variable is filled with a unique transaction identifier for use with the DdeAbandonTransaction function and the XTYP_XACT_COMPLETE transaction.

Return Values

If the function succeeds, the return value is a data handle that identifies the data for successful synchronous trans-

actions in which the client expects data from the server. The return value is nonzero for successful asynchronous transactions and for synchronous transactions in which the client does not expect data. The return value is zero for all unsuccessful transactions.

Errors

The DdeGetLastError function can be used to get the error code, which can be one of the following values:

DMLERR_ADVACKTIMEOUT

DMLERR_BUSY

DMLERR_DATAACKTIMEOUT

DMLERR_DLL_NOT_INITIALIZED

DMLERR_EXECACKTIMEOUT

DMLERR_INVALIDPARAMETER

DMLERR_MEMORY_ERROR

DMLERR_NO_CONV_ESTABLISHED

DMLERR_NO_ERROR

DMLERR_NOTPROCESSED

DMLERR_POKEACKTIMEOUT

DMLERR_POSTMSG_FAILED

DMLERR_REENTRANCY

DMLERR_SERVER_DIED

DMLERR_UNADVACKTIMEOUT

Remarks

When an application has finished using the data handle returned by DdeClientTransaction, the application should free the handle by calling the DdeFreeDataHandle function.

Transactions can be synchronous or asynchronous. During a synchronous transaction, DdeClientTransaction does not return until the transaction either completes successfully or fails. Synchronous transactions cause a client to enter a modal loop while waiting for various asynchronous events. Because of this, a client application can still respond to user input while waiting on a synchronous transaction, but the application cannot begin a second synchronous transaction because of the activity associated with the first. DdeClientTransaction fails if any instance of the same task has a synchronous transaction already in progress.

During an asynchronous transaction, DdeClientTransaction returns after the transaction has begun, passing a transaction identifier for reference. When the server's DDE callback function finishes processing an asynchro-

nous transaction, the system sends an XTYP_XACT_COMPLETE transaction to the client. This transaction provides the client with the results of the asynchronous transaction that it initiated by calling DdeClientTransaction. A client application can choose to abandon an asynchronous transaction by calling the DdeAbandonTransaction function.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.65 DdeCmpStringHandles

The DdeCmpStringHandles function compares the values of two string handles. The value of a string handle is not related to the case of the associated string.

```
DdeCmpStringHandles: procedure
(
    hsz1            :dword;
    hsz2            :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__DdeCmpStringHandles@8" );
```

Parameters

hsz1

[in] Handle to the first string.

hsz2

[in] Handle to the second string.

Return Values

The return value can be one of the following values:

Value	Meaning
–1	The value of hsz1 is either 0 or less than the value of hsz2.
0	The values of hsz1 and hsz2 are equal (both can be 0).
1	The value of hsz2 is either 0 or less than the value of hsz1.

Remarks

An application that must do a case-sensitive comparison of two string handles should compare the string handles directly. An application should use DdeCmpStringHandles for all other comparisons to preserve the case-insensitive nature of dynamic data exchange (DDE).

DdeCmpStringHandles cannot be used to sort string handles alphabetically.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.66 DdeConnect

The DdeConnect function establishes a conversation with a server application that supports the specified service name and topic name pair. If more than one such server exists, the system selects only one.

```
DdeConnect: procedure
(
        idInst      :dword;
        hszService  :dword;
        hszTopic    :dword;
    var pCC         :var
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__DdeConnect@16" );
```

Parameters

idInst

[in] Specifies the application instance identifier obtained by a previous call to the DdeInitialize function.

hszService

[in] Handle to the string that specifies the service name of the server application with which a conversation is to be established. This handle must have been created by a previous call to the DdeCreateStringHandle function. If this parameter is 0L, a conversation is established with any available server.

hszTopic

[in] Handle to the string that specifies the name of the topic on which a conversation is to be established. This handle must have been created by a previous call to DdeCreateStringHandle. If this parameter is 0L, a conversation on any topic supported by the selected server is established.

pCC

[in] Pointer to the CONVCONTEXT structure that contains conversation context information. If this parameter is NULL, the server receives the default CONVCONTEXT structure during the XTYP_CONNECT or XTYP_WILDCONNECT transaction.

Return Values

If the function succeeds, the return value is the handle to the established conversation.

If the function fails, the return value is 0L.

Errors

The DdeGetLastError function can be used to get the error code, which can be one of the following values:

DMLERR_DLL_NOT_INITIALIZED

DMLERR_INVALIDPARAMETER

DMLERR_NO_CONV_ESTABLISHED

DMLERR_NO_ERROR

Remarks

The client application cannot make assumptions regarding the server selected. If an instance-specific name is specified in the hszService parameter, a conversation is established with only the specified instance. Instance-specific service names are passed to an application's dynamic data exchange (DDE) callback function

during the XTYP_REGISTER and XTYP_UNREGISTER transactions.

All members of the default CONVCONTEXT structure are set to zero except cb, which specifies the size of the structure, and iCodePage, which specifies CP_WINANSI (the default code page) or CP_WINUNICODE, depending on whether the ANSI or Unicode version of the DdeInitialize function was called by the client application.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.67 DdeConnectList

The DdeConnectList function establishes a conversation with all server applications that support the specified service name and topic name pair. An application can also use this function to obtain a list of conversation handles by passing the function an existing conversation handle. The Dynamic Data Exchange Management Library removes the handles of any terminated conversations from the conversation list. The resulting conversation list contains the handles of all currently established conversations that support the specified service name and topic name.

```
DdeConnectList: procedure
(
        idInst      :dword;
        hszService  :dword;
        hszTopic    :dword;
        hConvList   :dword;
    var pCC         :var
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__DdeConnectList@20" );
```

Parameters

idInst

[in] Specifies the application instance identifier obtained by a previous call to the DdeInitialize function.

hszService

[in] Handle to the string that specifies the service name of the server application with which a conversation is to be established. If this parameter is 0L, the system attempts to establish conversations with all available servers that support the specified topic name.

hszTopic

[in] Handle to the string that specifies the name of the topic on which a conversation is to be established. This handle must have been created by a previous call to the DdeCreateStringHandle function. If this parameter is 0L, the system will attempt to establish conversations on all topics supported by the selected server (or servers).

hConvList

[in] Handle to the conversation list to be enumerated. This parameter should be 0L if a new conversation list is to be established.

pCC

[in] Pointer to the CONVCONTEXT structure that contains conversation-context information. If this parameter is NULL, the server receives the default CONVCONTEXT structure during the XTYP_CONNECT or XTYP_WILDCONNECT transaction.

Return Values

If the function succeeds, the return value is the handle to a new conversation list.

If the function fails, the return value is 0L. The handle to the old conversation list is no longer valid.

Errors

The DdeGetLastError function can be used to get the error code, which can be one of the following values:

DMLERR_DLL_NOT_INITIALIZED

DMLERR_INVALID_PARAMETER

DMLERR_NO_CONV_ESTABLISHED

DMLERR_NO_ERROR

DMLERR_SYS_ERROR

Remarks

An application must free the conversation list handle returned by the DdeConnectList function, regardless of whether any conversation handles within the list are active. To free the handle, an application can call DdeDisconnectList.

All members of the default CONVCONTEXT structure are set to zero except cb, specifying the size of the structure, and iCodePage, specifying CP_WINANSI (the default code page) or CP_WINUNICODE, depending on whether the ANSI or Unicode version of the DdeInitialize function was called by the client application.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.68 DdeCreateDataHandle

The DdeCreateDataHandle function creates a dynamic data exchange (DDE) object and fills the object with data from the specified buffer. A DDE application uses this function during transactions that involve passing data to the partner application.

```
DdeCreateDataHandle: procedure
(
        idInst      :dword;
    var pSrc        :var;
        cb          :dword;
        cbOff       :dword;
        hszItem     :dword;
        wFmt        :dword;
        afCmd       :dword
);
```

```
@stdcall;
@returns( "eax" );
@external( "__imp__DdeCreateDataHandle@28" );
```

Parameters

idInst

[in] Specifies the application instance identifier obtained by a previous call to the DdeInitialize function.

pSrc

[in] Pointer to a buffer that contains data to be copied to the DDE object. If this parameter is NULL, no data is copied to the object.

cb

[in] Specifies the amount of memory, in bytes, to copy from the buffer pointed to by pSrc. (include the terminating NULL, if the data is a string). If this parameter is zero, the pSrc parameter is ignored.

cbOff

[in] Specifies an offset, in bytes, from the beginning of the buffer pointed to by the pSrc parameter. The data beginning at this offset is copied from the buffer to the DDE object.

hszItem

[in] Handle to the string that specifies the data item corresponding to the DDE object. This handle must have been created by a previous call to the DdeCreateStringHandle function. If the data handle is to be used in an XTYP_EXECUTE transaction, this parameter must be 0L.

wFmt

[in] Specifies the standard clipboard format of the data.

afCmd

[in] Specifies the creation flags. This parameter can be HDATA_APPOWNED, which specifies that the server application calling the DdeCreateDataHandle function owns the data handle this function creates. This flag enables the application to share the data handle with other Dynamic Data Exchange Management Library (DDEML) applications rather than creating a separate handle to pass to each application. If this flag is specified, the application must eventually free the shared memory object associated with the handle by using the DdeFreeDataHandle function. If this flag is not specified, the handle becomes invalid in the application that created the handle after the data handle is returned by the application's DDE callback function or is used as a parameter in another DDEML function.

Return Values

If the function succeeds, the return value is a data handle.

If the function fails, the return value is 0L.

Errors

The DdeGetLastError function can be used to get the error code, which can be one of the following values:

DMLERR_DLL_NOT_INITIALIZED

DMLERR_INVALIDPARAMETER

DMLERR_MEMORY_ERROR

DMLERR_NO_ERROR

Remarks

Any unfilled locations in the DDE object are undefined.

After a data handle has been used as a parameter in another DDEML function or has been returned by a DDE callback function, the handle can be used only for read access to the DDE object identified by the handle.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.69 DdeCreateStringHandle

The DdeCreateStringHandle function creates a handle that identifies the specified string. A dynamic data exchange (DDE) client or server application can pass the string handle as a parameter to other Dynamic Data Exchange Management Library (DDEML) functions.

```
DdeCreateStringHandle: procedure
(
        idInst      :dword;
        psz         :string;
        iCodePage   :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__DdeCreateStringHandleA@12" );
```

Parameters

idInst

[in] Specifies the application instance identifier obtained by a previous call to the DdeInitialize function.

psz

[in] Pointer to a buffer that contains the null-terminated string for which a handle is to be created. This string can be up to 255 characters. The reason for this limit is that DDEML string management functions are implemented using global atoms.

iCodePage

[in] Specifies the code page used to render the string. This value should be either CP_WINANSI (the default code page) or CP_WINUNICODE, depending on whether the ANSI or Unicode version of DdeInitialize was called by the client application.

Return Values

If the function succeeds, the return value is a string handle.

If the function fails, the return value is 0L.

Errors

The DdeGetLastError function can be used to get the error code, which can be one of the following values:

DMLERR_INVALIDPARAMETER

DMLERR_NO_ERROR

DMLERR_SYS_ERROR

Remarks

The value of a string handle is not related to the case of the string it identifies.

When an application either creates a string handle or receives one in the callback function and then uses the DdeKeepStringHandle function to keep it, the application must free that string handle when it is no longer needed.

An instance-specific string handle cannot be mapped from string handle to string and back to string handle. This is shown in the following example, in which the DdeQueryString function creates a string from a string handle and DdeCreateStringHandle creates a string handle from that string, but the two handles are not the same:

DWORD idInst;

DWORD cb;

HSZ hszInst, hszNew;

PSZ pszInst;

DdeQueryString(idInst, hszInst, pszInst, cb, CP_WINANSI);

hszNew = DdeCreateStringHandle(idInst, pszInst, CP_WINANSI);

// hszNew != hszInst !

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.70 DdeDisconnect

The DdeDisconnect function terminates a conversation started by either the DdeConnect or DdeConnectList function and invalidates the specified conversation handle.

```
DdeDisconnect: procedure
(
    hConv           :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__DdeDisconnect@4" );
```

Parameters

hConv

[in/out] Handle to the active conversation to be terminated.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Errors

The DdeGetLastError function can be used to get the error code, which can be one of the following values:

DMLERR_DLL_NOT_INITIALIZED

DMLERR_NO_CONV_ESTABLISHED

DMLERR_NO_ERROR

Remarks

Any incomplete transactions started before calling DdeDisconnect are immediately abandoned. The XTYP_DISCONNECT transaction is sent to the dynamic data exchange (DDE) callback function of the partner in the conversation. Generally, only client applications must terminate conversations.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.71 DdeDisconnectList

The DdeDisconnectList function destroys the specified conversation list and terminates all conversations associated with the list.

```
DdeDisconnectList: procedure
(
    hConvList       :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__DdeDisconnectList@4" );
```

Parameters

hConvList

[in] Handle to the conversation list. This handle must have been created by a previous call to the DdeConnectList function.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Errors

The DdeGetLastError function can be used to get the error code, which can be one of the following values:

DMLERR_DLL_NOT_INITIALIZED

DMLERR_INVALIDPARAMETER

DMLERR_NO_ERROR

Remarks

An application can use the DdeDisconnect function to terminate individual conversations in the list.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.72 DdeEnableCallback

The DdeEnableCallback function enables or disables transactions for a specific conversation or for all conversations currently established by the calling application.

After disabling transactions for a conversation, the operating system places the transactions for that conversation in a transaction queue associated with the application. The application should reenable the conversation as soon as possible to avoid losing queued transactions.

```
DdeEnableCallback: procedure
(
    idInst          :dword;
    hConv           :dword;
    wCmd            :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__DdeEnableCallback@12" );
```

Parameters

idInst

[in] Specifies the application-instance identifier obtained by a previous call to the DdeInitialize function.

hConv

[in] Handle to the conversation to enable or disable. If this parameter is NULL, the function affects all conversations.

wCmd

[in] Specifies the function code. This parameter can be one of the following values.

Value	Meaning
EC_ENABLEALL	Enables all transactions for the specified conversation.
EC_ENABLEONE	Enables one transaction for the specified conversation.

EC_DISABLE — Disables all blockable transactions for the specified conversation.

A server application can disable the following transactions:

XTYP_ADVSTART

XTYP_ADVSTOP

XTYP_EXECUTE

XTYP_POKE

XTYP_REQUEST

A client application can disable the following transactions:

XTYP_ADVDATA

XTYP_XACT_COMPLETE

EC_QUERYWAITING — Determines whether any transactions are in the queue for the specified conversation.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

If the wCmd parameter is EC_QUERYWAITING, and the application transaction queue contains one or more unprocessed transactions that are not being processed, the return value is TRUE; otherwise, it is FALSE.

Errors

Use the DdeGetLastError function to retrieve the error code, which can be one of the following values:

DMLERR_DLL_NOT_INITIALIZED

DMLERR_NO_ERROR

DMLERR_INVALIDPARAMETER

Remarks

An application can disable transactions for a specific conversation by returning the CBR_BLOCK return code from its dynamic data exchange (DDE) callback function. When you reenable the conversation by using the DdeEnableCallback function, the operating system generates the same transaction that was in process when the conversation was disabled.

Using the EC_QUERYWAITING flag does not change the enable state of the conversation and does not cause transactions to be issued within the context of the call to DdeEnableCallback.

If DdeEnableCallback is called with EC_QUERYWAITING and the function returns a nonzero, an application should try to quickly allow message processing, return from its callback, or enable callbacks. Such a result does not guarantee that subsequent callbacks will be made. Calling DdeEnableCallback with EC_QUERYWAITING lets an application with blocked callbacks determine whether there are any transactions pending on the blocked conversation. Of course, even if such a call returns zero, an application should always process messages in a timely manner.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.73 DdeFreeDataHandle

The DdeFreeDataHandle function frees a dynamic data exchange (DDE) object and deletes the data handle associated with the object.

```
DdeFreeDataHandle: procedure
(
    hData           :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__DdeFreeDataHandle@4" );
```

Parameters

hData

[in/out] Handle to the DDE object to be freed. This handle must have been created by a previous call to the DdeCreateDataHandle function or returned by the DdeClientTransaction function.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Errors

The DdeGetLastError function can be used to get the error code, which can be one of the following values:

DMLERR_INVALIDPARAMETER

DMLERR_NO_ERROR

Remarks

An application must call DdeFreeDataHandle under the following circumstances:

- To free a DDE object that the application allocated by calling the DdeCreateDataHandle function if the object's data handle was never passed by the application to another Dynamic Data Exchange Management Library (DDEML) function
- To free a DDE object that the application allocated by specifying the HDATA_APPOWNED flag in a call to DdeCreateDataHandle
- To free a DDE object whose handle the application received from the DdeClientTransaction function

The system automatically frees an unowned object when its handle is returned by a DDE callback function or is used as a parameter in a DDEML function.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.74 DdeFreeStringHandle

The DdeFreeStringHandle function frees a string handle in the calling application.

```
DdeFreeStringHandle: procedure
(
    idInst          :dword;
    hsz             :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__DdeFreeStringHandle@8" );
```

Parameters

idInst

[in] Specifies the application instance identifier obtained by a previous call to the DdeInitialize function.

hsz

[in/out] Handle to the string handle to be freed. This handle must have been created by a previous call to the DdeCreateStringHandle function.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Remarks

An application can free string handles it creates with DdeCreateStringHandle but should not free those that the system passed to the application's dynamic data exchange (DDE) callback function or those returned in the CONVINFO structure by the DdeQueryConvInfo function.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.75 DdeGetData

The DdeGetData function copies data from the specified dynamic data exchange (DDE) object to the specified local buffer.

```
DdeGetData: procedure
(
        hData       :dword;
    var pDst        :var;
        cbMax       :dword;
        cbOff       :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__DdeGetData@16" );
```

Parameters

hData

[in] Handle to the DDE object that contains the data to copy.

pDst

[out] Pointer to the buffer that receives the data. If this parameter is NULL, the DdeGetData function returns the amount of data, in bytes, that would be copied to the buffer.

cbMax

[in] Specifies the maximum amount of data, in bytes, to copy to the buffer pointed to by the pDst parameter. Typically, this parameter specifies the length of the buffer pointed to by pDst.

cbOff

[in] Specifies an offset within the DDE object. Data is copied from the object beginning at this offset.

Return Values

If the pDst parameter points to a buffer, the return value is the size, in bytes, of the memory object associated with the data handle or the size specified in the cbMax parameter, whichever is lower.

If the pDst parameter is NULL, the return value is the size, in bytes, of the memory object associated with the data handle.

Errors

The DdeGetLastError function can be used to get the error code, which can be one of the following values:

DMLERR_DLL_NOT_INITIALIZED

DMLERR_INVALIDPARAMETER

DMLERR_NO_ERROR

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.76 DdeGetLastError

The DdeGetLastError function retrieves the most recent error code set by the failure of a Dynamic Data Exchange Management Library (DDEML) function and resets the error code to DMLERR_NO_ERROR.

```
DdeGetLastError: procedure
(
    idInst          :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__DdeGetLastError@4" );
```

Parameters

idInst

[in] Specifies the application instance identifier obtained by a previous call to the DdeInitialize function.

Return Values

If the function succeeds, the return value is the last error code, which can be one of the following values:

Value	Meaning
DMLERR_ADVACKTIMEOUT	A request for a synchronous advise transaction has timed out.
DMLERR_BUSY	The response to the transaction caused the DDE_FBUSY flag to be set.
DMLERR_DATAACKTIMEOUT	A request for a synchronous data transaction has timed out.
DMLERR_DLL_NOT_INITIALIZED	A DDEML function was called without first calling the DdeInitialize function, or an invalid instance identifier was passed to a DDEML function.
DMLERR_DLL_USAGE	An application initialized as APPCLASS_MONITOR has attempted to perform a dynamic data exchange (DDE) transaction, or an application initialized as APPCMD_CLIENTONLY has attempted to perform server transactions.
DMLERR_EXECACKTIMEOUT	A request for a synchronous execute transaction has timed out.
DMLERR_INVALIDPARAMETER	A parameter failed to be validated by the DDEML. Some of the possible causes follow: The application used a data handle initialized with a different item name handle than was required by the transaction. The application used a data handle that was initialized with a different clipboard data format than was required by the transaction. The application used a client-side conversation handle with a server-side function or vice versa. The application used a freed data handle or string handle. More than one instance of the application used the same object.
DMLERR_LOW_MEMORY	A DDEML application has created a prolonged race condition (in which the server application outruns the client), causing large amounts of memory to be consumed.
DMLERR_MEMORY_ERROR	A memory allocation has failed.
DMLERR_NO_CONV_ESTABLISHED	A client's attempt to establish a conversation has failed.
DMLERR_NOTPROCESSED	A transaction has failed.
DMLERR_POKEACKTIMEOUT	A request for a synchronous poke transaction has timed out.
DMLERR_POSTMSG_FAILED	An internal call to the PostMessage function has failed.

DMLERR_REENTRANCY	An application instance with a synchronous transaction already in progress attempted to initiate another synchronous transaction, or the DdeEnableCallback function was called from within a DDEML callback function.
DMLERR_SERVER_DIED	A server-side transaction was attempted on a conversation terminated by the client, or the server terminated before completing a transaction.
DMLERR_SYS_ERROR	An internal error has occurred in the DDEML.
DMLERR_UNADVACKTIMEOUT	A request to end an advise transaction has timed out.
DMLERR_UNFOUND_QUEUE_ID	An invalid transaction identifier was passed to a DDEML function. Once the application has returned from an XTYP_XACT_COMPLETE callback, the transaction identifier for that callback function is no longer valid.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.77 DdeImpersonateClient

The DdeImpersonateClient function impersonates a dynamic data exchange (DDE) client application in a DDE client conversation.

```
DdeImpersonateClient: procedure
(
    hConv       :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__DdeImpersonateClient@4" );
```

Parameters

hConv

[in] Handle to the DDE client conversation to be impersonated.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

Impersonation is the ability of a process to take on the security attributes of another process. When a client in a DDE conversation requests information from a DDE server, the server impersonates the client. When the server requests access to an object, the system verifies the access against the client's security attributes.

When the impersonation is complete, the server normally calls the RevertToSelf function.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Unsupported.

3.78 DdeInitialize

The DdeInitialize function registers an application with the Dynamic Data Exchange Management Library (DDEML). An application must call this function before calling any other DDEML function.

```
DdeInitialize: procedure
(
    var pidInst         :dword;
        _pfnCallback    :PFNCALLBACK;
        afCmd           :dword;
        ulRes           :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__DdeInitializeA@16" );
```

Parameters

pidInst

[in/out] Pointer to the application instance identifier. At initialization, this parameter should point to 0. If the function succeeds, this parameter points to the instance identifier for the application. This value should be passed as the idInst parameter in all other DDEML functions that require it. If an application uses multiple instances of the DDEML dynamic-link library (DLL), the application should provide a different callback function for each instance.

If pidInst points to a nonzero value, reinitialization of the DDEML is implied. In this case, pidInst must point to a valid application-instance identifier.

pfnCallback

[in] Pointer to the application-defined dynamic data exchange (DDE) callback function. This function processes DDE transactions sent by the system. For more information, see the DdeCallback callback function.

afCmd

[in] Specifies a set of APPCMD_, CBF_, and MF_ flags. The APPCMD_ flags provide special instructions to DdeInitialize. The CBF_ flags specify filters that prevent specific types of transactions from reaching the callback function. The MF_ flags specify the types of DDE activity that a DDE monitoring application monitors. Using these flags enhances the performance of a DDE application by eliminating unnecessary calls to the callback function.

This parameter can be one or more of the following values.

Value	Meaning
APPCLASS_MONITOR	Makes it possible for the application to monitor DDE activity in the system. This flag is for use by DDE monitoring applications. The application specifies the types of DDE activity to monitor by combining one or more monitor flags with the APPCLASS_MONITOR flag. For details, see the following Remarks section.
APPCLASS_STANDARD	Registers the application as a standard (nonmonitoring) DDEML application.

APPCMD_CLIENTONLY	Prevents the application from becoming a server in a DDE conversation. The application can only be a client. This flag reduces consumption of resources by the DDEML. It includes the functionality of the CBF_FAIL_ALLSVRXACTIONS flag.
APPCMD_FILTERINITS	Prevents the DDEML from sending XTYP_CONNECT and XTYP_WILDCONNECT transactions to the application until the application has created its string handles and registered its service names or has turned off filtering by a subsequent call to the DdeNameService or function. This flag is always in effect when an application calls for the first time, regardless of whether the application specifies the flag. On subsequent calls to , not specifying this flag turns off the application's service-name filters, but specifying it turns on the application's service name filters.
CBF_FAIL_ALLSVRXACTIONS	Prevents the callback function from receiving server transactions. The system returns DDE_FNOTPROCESSED to each client that sends a transaction to this application. This flag is equivalent to combining all CBF_FAIL_ flags.
CBF_FAIL_ADVISES	Prevents the callback function from receiving XTYP_ADVSTART and XTYP_ADVSTOP transactions. The system returns DDE_FNOTPROCESSED to each client that sends an XTYP_ADVSTART or XTYP_ADVSTOP transaction to the server.
CBF_FAIL_CONNECTIONS	Prevents the callback function from receiving XTYP_CONNECT and XTYP_WILDCONNECT transactions.
CBF_FAIL_EXECUTES	Prevents the callback function from receiving XTYP_EXECUTE transactions. The system returns DDE_FNOTPROCESSED to a client that sends an XTYP_EXECUTE transaction to the server.
CBF_FAIL_POKES	Prevents the callback function from receiving XTYP_POKE transactions. The system returns DDE_FNOTPROCESSED to a client that sends an XTYP_POKE transaction to the server.
CBF_FAIL_REQUESTS	Prevents the callback function from receiving XTYP_REQUEST transactions. The system returns DDE_FNOTPROCESSED to a client that sends an XTYP_REQUEST transaction to the server.
CBF_FAIL_SELFCONNECTIONS	Prevents the callback function from receiving XTYP_CONNECT transactions from the application's own instance. This flag prevents an application from establishing a DDE conversation with its own instance. An application should use this flag if it needs to communicate with other instances of itself but not with itself.
CBF_SKIP_ALLNOTIFICATIONS	Prevents the callback function from receiving any notifications. This flag is equivalent to combining all CBF_SKIP_ flags.
CBF_SKIP_CONNECT_CONFIRMS	Prevents the callback function from receiving XTYP_CONNECT_CONFIRM notifications.

CBF_SKIP_DISCONNECTS	Prevents the callback function from receiving XTYP_DISCONNECT notifications.
CBF_SKIP_REGISTRATIONS	Prevents the callback function from receiving XTYP_REGISTER notifications.
CBF_SKIP_UNREGISTRATIONS	Prevents the callback function from receiving XTYP_UNREGISTER notifications.
MF_CALLBACKS	Notifies the callback function whenever a transaction is sent to any DDE callback function in the system.
MF_CONV	Notifies the callback function whenever a conversation is established or terminated.
MF_ERRORS	Notifies the callback function whenever a DDE error occurs.
MF_HSZ_INFO	Notifies the callback function whenever a DDE application creates, frees, or increments the usage count of a string handle or whenever a string handle is freed as a result of a call to the DdeUninitialize function.
MF_LINKS	Notifies the callback function whenever an advise loop is started or ended.
MF_POSTMSGS	Notifies the callback function whenever the system or an application posts a DDE message.
MF_SENDMSGS	Notifies the callback function whenever the system or an application sends a DDE message.

ulRes

Reserved; must be set to zero.

Return Values

If the function succeeds, the return value is DMLERR_NO_ERROR.

If the function fails, the return value is one of the following values:

DMLERR_DLL_USAGE

DMLERR_INVALIDPARAMETER

DMLERR_SYS_ERROR

Remarks

An application that uses multiple instances of the DDEML must not pass DDEML objects between instances.

A DDE monitoring application should not attempt to perform DDE operations (establish conversations, issue transactions, and so on) within the context of the same application instance.

A synchronous transaction fails with a DMLERR_REENTRANCY error if any instance of the same task has a synchronous transaction already in progress.

The CBF_FAIL_ALLSVRACTIONS flag causes the DDEML to filter all server transactions and can be changed by a subsequent call to DdeInitialize. The APPCMD_CLIENTONLY flag prevents the DDEML from creating key resources for the server and cannot be changed by a subsequent call to DdeInitialize.

There is an ANSI version and a Unicode version of DdeInitialize. The version called determines the type of the window procedures used to control DDE conversations (ANSI or Unicode), and the default value for the iCodePage member of the CONVCONTEXT structure (CP_WINANSI or CP_WINUNICODE).

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.79 DdeKeepStringHandle

The DdeKeepStringHandle function increments the usage count associated with the specified handle. This function enables an application to save a string handle passed to the application's dynamic data exchange (DDE) callback function. Otherwise, a string handle passed to the callback function is deleted when the callback function returns. This function should also be used to keep a copy of a string handle referenced by the CONVINFO structure returned by the DdeQueryConvInfo function.

```
DdeKeepStringHandle: procedure
(
        idInst          :dword;
        hsz             :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__DdeKeepStringHandle@8" );
```

Parameters

idInst

[in] Specifies the application instance identifier obtained by a previous call to the DdeInitialize function.

hsz

[in/out] Handle to the string handle to be saved.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.80 DdeNameService

The DdeNameService function registers or unregisters the service names a dynamic data exchange (DDE) server supports. This function causes the system to send XTYP_REGISTER or XTYP_UNREGISTER transactions to other running Dynamic Data Exchange Management Library (DDEML) client applications.

A server application should call this function to register each service name that it supports and to unregister names it previously registered but no longer supports. A server should also call this function to unregister its service names just before terminating.

```
DdeNameService: procedure
(
    idInst          :dword;
    hsz1            :dword;
    hsz2            :dword;
    afCmd           :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__DdeNameService@16" );
```

Parameters

idInst

[in] Specifies the application instance identifier obtained by a previous call to the DdeInitialize function.

hsz1

[in] Handle to the string that specifies the service name the server is registering or unregistering. An application that is unregistering all of its service names should set this parameter to 0L.

hsz2

Reserved; should be set to 0L.

afCmd

[in] Specifies the service name options. This parameter can be one of the following values.

Value	Meaning
DNS_REGISTER	Registers the error code service name.
DNS_UNREGISTER	Unregisters the error code service name. If the hsz1 parameter is 0L, all service names registered by the server will be unregistered.
DNS_FILTERON	Turns on service name initiation filtering. The filter prevents a server from receiving XTYP_CONNECT transactions for service names it has not registered. This is the default setting for this filter. If a server application does not register any service names, the application cannot receive XTYP_WILDCONNECT transactions.
DNS_FILTEROFF	Turns off service name initiation filtering. If this flag is specified, the server receives an XTYP_CONNECT transaction whenever another DDE application calls the DdeConnect function, regardless of the service name.

Return Values

If the function succeeds, it returns a nonzero value. That value is not a true HDDEDATA value, merely a Boolean indicator of success. The function is typed HDDEDATA to allow for possible future expansion of the function and a more sophisticated return value.

If the function fails, the return value is 0L.

Errors

The DdeGetLastError function can be used to get the error code, which can be one of the following values:

DMLERR_DLL_NOT_INITIALIZED

DMLERR_DLL_USAGE

DMLERR_INVALIDPARAMETER

DMLERR_NO_ERROR

Remarks

The service name identified by the hsz1 parameter should be a base name (that is, the name should contain no instance-specific information). The system generates an instance-specific name and sends it along with the base name during the XTYP_REGISTER and XTYP_UNREGISTER transactions. The receiving applications can then connect to the specific application instance.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.81 DdePostAdvise

The DdePostAdvise function causes the system to send an XTYP_ADVREQ transaction to the calling (server) application's dynamic data exchange (DDE) callback function for each client with an active advise loop on the specified topic and item. A server application should call this function whenever the data associated with the topic name or item name pair changes.

```
DdePostAdvise: procedure
(
    idInst          :dword;
    hszTopic        :dword;
    hszItem         :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__DdePostAdvise@12" );
```

Parameters

idInst

[in] Specifies the application instance identifier obtained by a previous call to the DdeInitialize function.

hszTopic

[in] Handle to a string that specifies the topic name. To send notifications for all topics with active advise loops, an application can set this parameter to 0L.

hszItem

[in] Handle to a string that specifies the item name. To send notifications for all items with active advise loops, an application can set this parameter to 0L.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Errors

The DdeGetLastError function can be used to get the error code, which can be one of the following values:

DMLERR_DLL_NOT_INITIALIZED

DMLERR_DLL_USAGE

DMLERR_NO_ERROR

Remarks

A server that has nonenumerable topics or items should set the hszTopic and hszItem parameters to NULL so that the system generates transactions for all active advise loops. The server's DDE callback function returns NULL for any advise loops that must not be updated.

If a server calls DdePostAdvise with a topic, item, and format name set that includes the set currently being handled in an XTYP_ADVREQ callback, a stack overflow can result.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.82 DdeQueryConvInfo

The DdeQueryConvInfo function retrieves information about a dynamic data exchange (DDE) transaction and about the conversation in which the transaction takes place.

```
DdeQueryConvInfo: procedure
(
        hConv           :dword;
        idTransaction   :dword;
    var pConvInfo       :var
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__DdeQueryConvInfo@12" );
```

Parameters

hConv

[in] Handle to the conversation.

idTransaction

[in] Specifies the transaction. For asynchronous transactions, this parameter should be a transaction identifier returned by the DdeClientTransaction function. For synchronous transactions, this parameter should be QID_SYNC.

pConvInfo

[in/out] Pointer to the CONVINFO structure that receives information about the transaction and conversation. The cb member of the CONVINFO structure must specify the length of the buffer allocated for the structure.

Return Values

If the function succeeds, the return value is the number of bytes copied into the CONVINFO structure.

If the function fails, the return value is FALSE.

Errors

The DdeGetLastError function can be used to get the error code, which can be one of the following values:

DMLERR_DLL_NOT_INITIALIZED

DMLERR_NO_CONV_ESTABLISHED

DMLERR_NO_ERROR

DMLERR_UNFOUND_QUEUE_ID

Remarks

An application should not free a string handle referenced by the CONVINFO structure. If an application must use one of these string handles, it should call the DdeKeepStringHandle function to create a copy of the handle.

If the idTransaction parameter is set to QID_SYNC, the hUser member of the CONVINFO structure is associated with the conversation and can be used to hold data associated with the conversation. If idTransaction is the identifier of an asynchronous transaction, the hUser member is associated only with the current transaction and is valid only for the duration of the transaction.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.83 DdeQueryNextServer

The DdeQueryNextServer function retrieves the next conversation handle in the specified conversation list.

```
DdeQueryNextServer: procedure
(
        hConvList       :dword;
        hConvPrev       :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__DdeQueryNextServer@8" );
```

Parameters

hConvList

[in] Handle to the conversation list. This handle must have been created by a previous call to the DdeConnectList function.

hConvPrev

[in] Handle to the conversation handle previously returned by this function. If this parameter is 0L, the function returns the first conversation handle in the list.

Return Values

If the list contains any more conversation handles, the return value is the next conversation handle in the list; otherwise, it is 0L.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.84 DdeQueryString

The DdeQueryString function copies text associated with a string handle into a buffer.

```
DdeQueryString: procedure
```

```
(
    idInst          :dword;
    hsz             :dword;
    psz             :string;
    pcchMax3        :dword;
    iCodePage       :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__DdeQueryStringA@20" );
```

Parameters

idInst

[in] Specifies the application instance identifier obtained by a previous call to the DdeInitialize function.

hsz

[in] Handle to the string to copy. This handle must have been created by a previous call to the DdeCreateString-Handle function.

psz

[in/out] Pointer to a buffer that receives the string. To obtain the length of the string, this parameter should be set to NULL.

cchMax

[in] Specifies the length, in TCHARs, of the buffer pointed to by the psz parameter. For the ANSI version of the function, this is the number of bytes; for the Unicode version, this is the number of characters. If the string is longer than (cchMax – 1), it will be truncated. If the psz parameter is set to NULL, this parameter is ignored.

iCodePage

[in] Specifies the code page used to render the string. This value should be either CP_WINANSI or CP_WINUNICODE.

Return Values

If the psz parameter specified a valid pointer, the return value is the length, in TCHARs, of the returned text (not including the terminating null character). If the psz parameter specified a NULL pointer, the return value is the length of the text associated with the hsz parameter (not including the terminating null character). If an error occurs, the return value is 0L.

Remarks

The string returned in the buffer is always null-terminated. If the string is longer than (cchMax – 1), only the first (cchMax – 1) characters of the string are copied.

If the psz parameter is NULL, the DdeQueryString function obtains the length, in bytes, of the string associated with the string handle. The length does not include the terminating null character.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.85 DdeReconnect

The DdeReconnect function enables a client Dynamic Data Exchange Management Library (DDEML) application to attempt to reestablish a conversation with a service that has terminated a conversation with the client. When the conversation is reestablished, the DDEML attempts to reestablish any preexisting advise loops.

```
DdeReconnect: procedure
(
    hConv           :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__DdeReconnect@4" );
```

Parameters

hConv

[in] Handle to the conversation to be reestablished. A client must have obtained the conversation handle by a previous call to the DdeConnect function or from an XTYP_DISCONNECT transaction.

Return Values

If the function succeeds, the return value is the handle to the reestablished conversation.

If the function fails, the return value is 0L.

Errors

The DdeGetLastError function can be used to get the error code, which can be one of the following values:

DMLERR_DLL_NOT_INITIALIZED

DMLERR_INVALIDPARAMETER

DMLERR_NO_CONV_ESTABLISHED

DMLERR_NO_ERROR

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.86 DdeSetQualityOfService

The DdeSetQualityOfService function specifies the quality of service (QOS) a raw DDE application desires for future DDE conversations it initiates. The specified QOS applies to any conversations started while those settings are in place. A DDE conversation's quality of service lasts for the duration of the conversation; calls to the DdeSetQualityOfService function during a conversation do not affect that conversation's QOS.

```
DdeSetQualityOfService: procedure
(
        hwndClient      :dword;
    var pqosNew         :SECURITY_QUALITY_OF_SERVICE;
    var pqosPrev        :SECURITY_QUALITY_OF_SERVICE
```

```
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__DdeSetQualityOfService@12" );
```

Parameters

hwndClient

[in] Handle to the DDE client window that specifies the source of WM_DDE_INITIATE messages a client will send to start DDE conversations.

pqosNew

[in] Pointer to a SECURITY_QUALITY_OF_SERVICE structure for the desired quality of service values.

pqosPrev

[out] Pointer to a SECURITY_QUALITY_OF_SERVICE structure that receives the previous quality of service values associated with the window identified by hwndClient.

This parameter is optional. If an application has no interest in hwndClient's previous QOS values, it should set pqosPrev to NULL.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

If a quality of service has not been specified for a client window, hwndClient, prior to sending a WM_DDE_INITIATE with the wParam set to hwndClient, the system uses the following default quality of service values for the client window:

{

Length = sizeof(SECURITY_QUALITY_OF_SERVICE);

ImpersonationLevel = SecurityImpersonation;

ContextTrackingMode = SECURITY_STATIC_TRACKING;

EffectiveOnly = TRUE;

}

Use the DdeSetQualityOfService function to associate a different quality of service with the client window. After you change the quality of service, the new settings affect any subsequent conversations that are started. Once an application starts a DDE conversation using a particular quality of service value, it must terminate the conversation and restart the conversation in order to have a different value take effect.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Unsupported.

3.87 DdeSetUserHandle

The DdeSetUserHandle function associates an application-defined value with a conversation handle or a transaction identifier. This is useful for simplifying the processing of asynchronous transactions. An application can use

the DdeQueryConvInfo function to retrieve this value.

```
DdeSetUserHandle: procedure
(
    hConv           :dword;
    id              :dword;
    hUser           :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__DdeSetUserHandle@12" );
```

Parameters

hConv

[in] Handle to the conversation.

id

[in] Specifies the transaction identifier to associate with the value specified by the hUser parameter. An application should set this parameter to QID_SYNC to associate hUser with the conversation identified by the hConv parameter.

hUser

[in] Value to associate with the conversation handle.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Errors

The DdeGetLastError function can be used to get the error code, which can be one of the following values:

DMLERR_DLL_NOT_INITIALIZED

DMLERR_INVALIDPARAMETER

DMLERR_NO_ERROR

DMLERR_UNFOUND_QUEUE_ID

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.88 DdeUnaccessData

The DdeUnaccessData function unaccesses a dynamic data exchange (DDE) object. An application must call this function after it has finished accessing the object.

```
DdeUnaccessData: procedure
(
    hData           :dword
```

```
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__DdeUnaccessData@4" );
```

Parameters

hData

[in] Handle to the DDE object.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Errors

The DdeGetLastError function can be used to get the error code, which can be one of the following values:

DMLERR_DLL_NOT_INITIALIZED

DMLERR_INVALIDPARAMETER

DMLERR_NO_ERROR

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.89 DdeUninitialize

The DdeUninitialize function frees all Dynamic Data Exchange Management Library (DDEML) resources associated with the calling application.

```
DdeUninitialize: procedure
(
    idInst          :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__DdeUninitialize@4" );
```

Parameters

idInst

[in] Specifies the application instance identifier obtained by a previous call to the DdeInitialize function.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Remarks

DdeUninitialize terminates any conversations currently open for the application.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.90 DefDlgProc

The DefDlgProc function calls the default dialog box window procedure to provide default processing for any window messages that a dialog box with a private window class does not process.

```
DefDlgProc: procedure
(
    hDlg            :dword;
    _Msg            :dword;
    _wParam         :dword;
    _lParam         :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__DefDlgProcA@16" );
```

Parameters

hDlg

[in] Handle to the dialog box.

Msg

[in] Specifies the message.

wParam

[in] Specifies additional message-specific information.

lParam

[in] Specifies additional message-specific information.

Return Values

The return value specifies the result of the message processing and depends on the message sent.

Remarks

The DefDlgProc function is the window procedure for the predefined class of dialog box. This procedure provides internal processing for the dialog box by forwarding messages to the dialog box procedure and carrying out default processing for any messages that the dialog box procedure returns as FALSE. Applications that create custom window procedures for their custom dialog boxes often use DefDlgProc instead of the DefWindowProc function to carry out default message processing.

Applications create custom dialog box classes by filling a WNDCLASS structure with appropriate information and registering the class with the RegisterClass function. Some applications fill the structure by using the GetClassInfo function, specifying the name of the predefined dialog box. In such cases, the applications modify at least the lpszClassName member before registering. In all cases, the cbWndExtra member of WNDCLASS for a custom dialog box class must be set to at least DLGWINDOWEXTRA.

The DefDlgProc function must not be called by a dialog box procedure; doing so results in recursive execution.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.91 DefFrameProc

The DefFrameProc function provides default processing for any window messages that the window procedure of a multiple document interface (MDI) frame window does not process. All window messages that are not explicitly processed by the window procedure must be passed to the DefFrameProc function, not the DefWindowProc function.

```
DefFrameProc: procedure
(
    hWnd            :dword;
    hWndMDIClient   :dword;
    uMsg            :dword;
    _wParam         :dword;
    _lParam         :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__DefFrameProcA@20" );
```

Parameters

hWnd

[in] Handle to the MDI frame window.

hWndMDIClient

[in] Handle to the MDI client window.

uMsg

[in] Specifies the message to be processed.

wParam

[in] Specifies additional message-specific information.

lParam

[in] Specifies additional message-specific information.

Return Values

The return value specifies the result of the message processing and depends on the message. If the hWndMDIClient parameter is NULL, the return value is the same as for the DefWindowProc function.

Remarks

When an application's window procedure does not handle a message, it typically passes the message to the DefWindowProc function to process the message. MDI applications use the DefFrameProc and DefMDIChildProc functions instead of DefWindowProc to provide default message processing. All messages that an application would usually pass to DefWindowProc (such as nonclient messages and the WM_SETTEXT message) should be passed to DefFrameProc instead. The DefFrameProc function also handles the following messages.

Message	Response

WM_COMMAND	Activates the MDI child window that the user chooses. This message is sent when the user chooses an MDI child window from the window menu of the MDI frame window. The window identifier accompanying this message identifies the MDI child window to be activated.
WM_MENUCHAR	Opens the window menu of the active MDI child window when the user presses the ALT+ – (minus) key combination.
WM_SETFOCUS	Passes the keyboard focus to the MDI client window, which in turn passes it to the active MDI child window.
WM_SIZE	Resizes the MDI client window to fit in the new frame window's client area. If the frame window procedure sizes the MDI client window to a different size, it should not pass the message to the DefWindowProc function.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.92 DefMDIChildProc

The DefMDIChildProc function provides default processing for any window message that the window procedure of a multiple document interface (MDI) child window does not process. A window message not processed by the window procedure must be passed to the DefMDIChildProc function, not to the DefWindowProc function.

```
DefMDIChildProc: procedure
(
    hWnd            :dword;
    uMsg            :dword;
    _wParam         :dword;
    _lParam         :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__DefMDIChildProcA@16" );
```

Parameters

hWnd

[in] Handle to the MDI child window.

uMsg

[in] Specifies the message to be processed.

wParam

[in] Specifies additional message-specific information.

lParam

[in] Specifies additional message-specific information.

Return Values

The return value specifies the result of the message processing and depends on the message.

Remarks

The DefMDIChildProc function assumes that the parent window of the MDI child window identified by the

hWnd parameter was created with the MDICLIENT class.

When an application's window procedure does not handle a message, it typically passes the message to the DefWindowProc function to process the message. MDI applications use the DefFrameProc and DefMDIChildProc functions instead of DefWindowProc to provide default message processing. All messages that an application would usually pass to DefWindowProc (such as nonclient messages and the WM_SETTEXT message) should be passed to DefMDIChildProc instead. In addition, DefMDIChildProc also handles the following messages.

Message	Response
WM_CHILDACTIVATE	Performs activation processing when MDI child windows are sized, moved, or displayed. This message must be passed.
WM_GETMINMAXINFO	Calculates the size of a maximized MDI child window, based on the current size of the MDI client window.
WM_MENUCHAR	Passes the message to the MDI frame window.
WM_MOVE	Recalculates MDI client scroll bars if they are present.
WM_SETFOCUS	Activates the child window if it is not the active MDI child window.
WM_SIZE	Performs operations necessary for changing the size of a window, especially for maximizing or restoring an MDI child window. Failing to pass this message to the **function produces highly undesirable results.**
WM_SYSCOMMAND	Handles window menu commands: SC_NEXTWINDOW, SC_PREVWINDOW, SC_MOVE, SC_SIZE, and SC_MAXIMIZE.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.93 DefWindowProc

The DefWindowProc function calls the default window procedure to provide default processing for any window messages that an application does not process. This function ensures that every message is processed. DefWindowProc is called with the same parameters received by the window procedure.

```
DefWindowProc: procedure
(
    hWnd        :dword;
    _Msg        :dword;
    _wParam     :dword;
    _lParam     :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__DefWindowProcA@16" );
```

Parameters

hWnd

[in] Handle to the window procedure that received the message.

Msg

[in] Specifies the message.

wParam

[in] Specifies additional message information. The content of this parameter depends on the value of the Msg parameter.

lParam

[in] Specifies additional message information. The content of this parameter depends on the value of the Msg parameter.

Return Values

The return value is the result of the message processing and depends on the message.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.94 DeferWindowPos

The DeferWindowPos function updates the specified multiple-window – position structure for the specified window. The function then returns a handle to the updated structure. The EndDeferWindowPos function uses the information in this structure to change the position and size of a number of windows simultaneously. The BeginDeferWindowPos function creates the structure.

```
DeferWindowPos: procedure
(
    hWinPosInfo     :dword;
    hWnd            :dword;
    hWndInsertAfter :dword;
    x               :dword;
    y               :dword;
    cx              :dword;
    cy              :dword;
    uFlags          :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__DeferWindowPos@32" );
```

Parameters

hWinPosInfo

[in] Handle to a multiple-window – position structure that contains size and position information for one or more windows. This structure is returned by BeginDeferWindowPos or by the most recent call to DeferWindowPos.

hWnd

[in] Handle to the window for which update information is stored in the structure. All windows in a multiple-window – position structure must have the same parent.

hWndInsertAfter

[in] Handle to the window that precedes the positioned window in the Z order. This parameter must be a window handle or one of the following values.

Value	Meaning
HWND_BOTTOM	Places the window at the bottom of the Z order. If the hWnd parameter identifies a topmost window, the window loses its topmost status and is placed at the bottom of all other windows.
HWND_NOTOPMOST	Places the window above all non-topmost windows (that is, behind all topmost windows). This flag has no effect if the window is already a non-topmost window.
HWND_TOP	Places the window at the top of the Z order.
HWND_TOPMOST	Places the window above all non-topmost windows. The window maintains its topmost position even when it is deactivated.

This parameter is ignored if the SWP_NOZORDER flag is set in the uFlags parameter.

x

[in] Specifies the x-coordinate of the window's upper-left corner.

y

[in] Specifies the y-coordinate of the window's upper-left corner.

cx

[in] Specifies the window's new width, in pixels.

cy

[in] Specifies the window's new height, in pixels.

uFlags

[in] Specifies a combination of the following values that affect the size and position of the window.

Value	Meaning
SWP_DRAWFRAME	Draws a frame (defined in the window's class description) around the window.
SWP_FRAMECHANGED	Sends a WM_NCCALCSIZE message to the window, even if the window's size is not being changed. If this flag is not specified, WM_NCCALCSIZE is sent only when the window's size is being changed.
SWP_HIDEWINDOW	Hides the window.
SWP_NOACTIVATE	Does not activate the window. If this flag is not set, the window is activated and moved to the top of either the topmost or non-topmost group (depending on the setting of the hWndInsertAfter parameter).
SWP_NOCOPYBITS	Discards the entire contents of the client area. If this flag is not specified, the valid contents of the client area are saved and copied back into the client area after the window is sized or repositioned.
SWP_NOMOVE	Retains the current position (ignores the X and Y parameters).
SWP_NOOWNERZORDER	Does not change the owner window's position in the Z order.
SWP_NOREDRAW	Does not redraw changes. If this flag is set, no repainting of any kind occurs. This applies to the client area, the nonclient area (including the title bar and scroll bars), and any part of the parent window uncovered as a result of the window being moved. When this flag is set, the application must explicitly invalidate or redraw any parts of the window and parent window that need redrawing.
SWP_NOREPOSITION	Same as the SWP_NOOWNERZORDER flag.

SWP_NOSENDCHANGING	Prevents the window from receiving the WM_WINDOWPOSCHANGING message.
SWP_NOSIZE	Retains the current size (ignores the cx and cy parameters).
SWP_NOZORDER	Retains the current Z order (ignores the hWndInsertAfter parameter).
SWP_SHOWWINDOW	Displays the window.

Return Values

The return value identifies the updated multiple-window – position structure. The handle returned by this function may differ from the handle passed to the function. The new handle that this function returns should be passed during the next call to the DeferWindowPos or EndDeferWindowPos function.

If insufficient system resources are available for the function to succeed, the return value is NULL. To get extended error information, call GetLastError.

Remarks

If a call to DeferWindowPos fails, the application should abandon the window-positioning operation and not call EndDeferWindowPos.

If SWP_NOZORDER is not specified, the system places the window identified by the hWnd parameter in the position following the window identified by the hWndInsertAfter parameter. If hWndInsertAfter is NULL or HWND_TOP, the system places the hWnd window at the top of the Z order. If hWndInsertAfter is set to HWND_BOTTOM, the system places the hWnd window at the bottom of the Z order.

All coordinates for child windows are relative to the upper-left corner of the parent window's client area.

A window can be made a topmost window either by setting hWndInsertAfter to the HWND_TOPMOST flag and ensuring that the SWP_NOZORDER flag is not set, or by setting the window's position in the Z order so that it is above any existing topmost windows. When a non-topmost window is made topmost, its owned windows are also made topmost. Its owners, however, are not changed.

If neither the SWP_NOACTIVATE nor SWP_NOZORDER flag is specified (that is, when the application requests that a window be simultaneously activated and its position in the Z order changed), the value specified in hWndInsertAfter is used only in the following circumstances:

- Neither the HWND_TOPMOST nor HWND_NOTOPMOST flag is specified in hWndInsertAfter.
- The window identified by hWnd is not the active window.

An application cannot activate an inactive window without also bringing it to the top of the Z order. An application can change an activated window's position in the Z order without restrictions, or it can activate a window and then move it to the top of the topmost or non-topmost windows.

A topmost window is no longer topmost if it is repositioned to the bottom (HWND_BOTTOM) of the Z order or after any non-topmost window. When a topmost window is made non-topmost, its owners and its owned windows are also made non-topmost windows.

A non-topmost window may own a topmost window, but not vice versa. Any window (for example, a dialog box) owned by a topmost window is itself made a topmost window to ensure that all owned windows stay above their owner.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.95 DeleteMenu

The DeleteMenu function deletes an item from the specified menu. If the menu item opens a menu or submenu, this function destroys the handle to the menu or submenu and frees the memory used by the menu or submenu.

```
DeleteMenu: procedure
(
    hMenu           :dword;
    uPosition       :dword;
    uflags          :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__DeleteMenu@12" );
```

Parameters

hMenu

[in] Handle to the menu to be changed.

uPosition

[in] Specifies the menu item to be deleted, as determined by the uFlags parameter.

uFlags

[in] Specifies how the uPosition parameter is interpreted. This parameter must be one of the following values.

Value	Meaning
MF_BYCOMMAND	Indicates that uPosition gives the identifier of the menu item. The MF_BYCOMMAND flag is the default flag if neither the MF_BYCOMMAND nor MF_BYPOSITION flag is specified.
MF_BYPOSITION	Indicates that uPosition gives the zero-based relative position of the menu item.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

The application must call the DrawMenuBar function whenever a menu changes, whether or not the menu is in a displayed window.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.96 DestroyAcceleratorTable

The DestroyAcceleratorTable function destroys an accelerator table. Before an application closes, it must use this function to destroy each accelerator table that it created by using the CreateAcceleratorTable function.

```
DestroyAcceleratorTable: procedure
(
    hAccel          :dword
```

```
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__DestroyAcceleratorTable@4" );
```

Parameters

hAccel

[in] Handle to the accelerator table to destroy. This handle must have been created by a call to the CreateAcceleratorTable function.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.97 DestroyCaret

The DestroyCaret function destroys the caret's current shape, frees the caret from the window, and removes the caret from the screen.

If the caret shape is based on a bitmap, DestroyCaret does not free the bitmap.

```
DestroyCaret: procedure;
    @stdcall;
    @returns( "eax" );
    @external( "__imp__DestroyCaret@0" );
```

Parameters

This function has no parameters.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

DestroyCaret destroys the caret only if a window in the current task owns the caret. If a window that is not in the current task owns the caret, DestroyCaret does nothing and returns FALSE.

The system provides one caret per queue. A window should create a caret only when it has the keyboard focus or is active. The window should destroy the caret before losing the keyboard focus or becoming inactive.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.98 DestroyCursor

The DestroyCursor function destroys a cursor and frees any memory the cursor occupied. Do not use this function to destroy a shared cursor.

```
DestroyCursor: procedure
(
    hCursor         :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__DestroyCursor@4" );
```

Parameters

hCursor

[in] Handle to the cursor to be destroyed. The cursor must not be in use.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

The DestroyCursor function destroys a nonshared cursor. Do not use this function to destroy a shared cursor. A shared cursor is valid as long as the module from which it was loaded remains in memory. The following functions obtain a shared cursor:

- LoadCursor
- LoadCursorFromFile
- LoadImage (if you use the LR_SHARED flag)
- CopyImage (if you use the LR_COPYRETURNORG flag and the hImage parameter is a shared cursor)

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.99 DestroyIcon

The DestroyIcon function destroys an icon and frees any memory the icon occupied.

```
DestroyIcon: procedure
(
    hIcon           :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__DestroyIcon@4" );
```

Parameters

hIcon

[in] Handle to the icon to be destroyed. The icon must not be in use.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

It is only necessary to call DestroyIcon for icons and cursors created with the CreateIconIndirect and the CopyIcon functions. Do not use this function to destroy a shared icon. A shared icon is valid as long as the module from which it was loaded remains in memory. The following functions obtain a shared icon:

- LoadIcon
- LoadImage (if you use the LR_SHARED flag)
- CopyImage (if you use the LR_COPYRETURNORG flag and the hImage parameter is a shared icon)

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.100 DestroyMenu

The DestroyMenu function destroys the specified menu and frees any memory that the menu occupies.

```
DestroyMenu: procedure
(
    hMenu           :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__DestroyMenu@4" );
```

Parameters

hMenu

[in] Handle to the menu to be destroyed.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

Before closing, an application must use the DestroyMenu function to destroy a menu not assigned to a window. A menu that is assigned to a window is automatically destroyed when the application closes.

DestroyMenu is recursive, that is, it will destroy the menu and all its submenus.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.101 DestroyWindow

The DestroyWindow function destroys the specified window. The function sends WM_DESTROY and WM_NCDESTROY messages to the window to deactivate it and remove the keyboard focus from it. The function also destroys the window's menu, flushes the thread message queue, destroys timers, removes clipboard ownership, and breaks the clipboard viewer chain (if the window is at the top of the viewer chain).

If the specified window is a parent or owner window, DestroyWindow automatically destroys the associated child or owned windows when it destroys the parent or owner window. The function first destroys child or owned windows, and then it destroys the parent or owner window.

DestroyWindow also destroys modeless dialog boxes created by the CreateDialog function.

```
DestroyWindow: procedure
(
    hWnd            :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__DestroyWindow@4" );
```

Parameters

hWnd

[in] Handle to the window to be destroyed.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

A thread cannot use DestroyWindow to destroy a window created by a different thread.

If the window being destroyed is a child window that does not have the WS_EX_NOPARENTNOTIFY style, a WM_PARENTNOTIFY message is sent to the parent.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.102 DialogBoxIndirectParam

The DialogBoxIndirectParam function creates a modal dialog box from a dialog box template in memory. Before displaying the dialog box, the function passes an application-defined value to the dialog box procedure as the lParam parameter of the WM_INITDIALOG message. An application can use this value to initialize dialog box controls.

```
DialogBoxIndirectParam: procedure
(
        hInstance       :dword;
    var hDialogTemplate :CDLGTEMPLATE;
        hWndParent      :dword;
        lpDialogProc    :DLGPROC;
        dwInitParam     :dword
```

```
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__DialogBoxIndirectParamA@20" );
```

Parameters

hInstance

[in] Handle to the module that creates the dialog box.

hDialogTemplate

[in] Pointer to a global memory object that contains the template that DialogBoxIndirectParam uses to create the dialog box. A dialog box template consists of a header that describes the dialog box, followed by one or more additional blocks of data that describe each of the controls in the dialog box. The template can use either the standard format or the extended format.

In a standard template for a dialog box, the header is a DLGTEMPLATE structure followed by additional variable-length arrays. The data for each control consists of a DLGITEMTEMPLATE structure followed by additional variable-length arrays.

In an extended template for a dialog box, the header uses the DLGTEMPLATEEX format and the control definitions use the DLGITEMTEMPLATEEX format.

hWndParent

[in] Handle to the window that owns the dialog box.

lpDialogFunc

[in] Pointer to the dialog box procedure. For more information about the dialog box procedure, see DialogProc.

dwInitParam

[in] Specifies the value to pass to the dialog box in the lParam parameter of the WM_INITDIALOG message.

Return Values

If the function succeeds, the return value is the nResult parameter specified in the call to the EndDialog function that was used to terminate the dialog box.

If the function fails because the hWndParent parameter is invalid, the return value is zero. The function returns zero in this case for compatibility with previous versions of Windows. If the function fails for any other reason, the return value is –1. To get extended error information, call GetLastError.

Remarks

The DialogBoxIndirectParam function uses the CreateWindowEx function to create the dialog box. DialogBoxIndirectParam then sends a WM_INITDIALOG message to the dialog box procedure. If the template specifies the DS_SETFONT or DS_SHELLFONT style, the function also sends a WM_SETFONT message to the dialog box procedure. The function displays the dialog box (regardless of whether the template specifies the WS_VISIBLE style), disables the owner window, and starts its own message loop to retrieve and dispatch messages for the dialog box.

When the dialog box procedure calls the EndDialog function, DialogBoxIndirectParam destroys the dialog box, ends the message loop, enables the owner window (if previously enabled), and returns the nResult parameter specified by the dialog box procedure when it called EndDialog.

In a standard dialog box template, the DLGTEMPLATE structure and each of the DLGITEMTEMPLATE structures must be aligned on DWORD boundaries. The creation data array that follows a DLGITEMTEMPLATE structure must also be aligned on a DWORD boundary. All of the other variable-length arrays in the template must be aligned on WORD boundaries.

In an extended dialog box template, the DLGTEMPLATEEX header and each of the DLGITEMTEMPLATEEX

control definitions must be aligned on DWORD boundaries. The creation data array, if any, that follows a DLGITEMTEMPLATEEX structure must also be aligned on a DWORD boundary. All of the other variable-length arrays in the template must be aligned on WORD boundaries.

All character strings in the dialog box template, such as titles for the dialog box and buttons, must be Unicode strings. To create code that works on both Windows 95/98 and Windows NT/Windows 2000, use the MultiByteToWideChar function to generate these Unicode strings.

Windows 95/98: The system can support a maximum of 255 controls per dialog box template. To place more than 255 controls in a dialog box, create the controls in the WM_INITDIALOG message handler rather than placing them in the template.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.103 DialogBoxParam

The DialogBoxParam function creates a modal dialog box from a dialog box template resource. Before displaying the dialog box, the function passes an application-defined value to the dialog box procedure as the lParam parameter of the WM_INITDIALOG message. An application can use this value to initialize dialog box controls.

```
DialogBoxParam: procedure
(
    hInstance       :dword;
    lpTemplateName  :string;
    hWndParent      :dword;
    lpDialogFunc    :DLGPROC;
    dwInitParam     :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__DialogBoxParamA@20" );
```

Parameters

hInstance

[in] Handle to the module whose executable file contains the dialog box template.

lpTemplateName

[in] Specifies the dialog box template. This parameter is either the pointer to a null-terminated character string that specifies the name of the dialog box template or an integer value that specifies the resource identifier of the dialog box template. If the parameter specifies a resource identifier, its high-order word must be zero and its low-order word must contain the identifier. You can use the MAKEINTRESOURCE macro to create this value.

hWndParent

[in] Handle to the window that owns the dialog box.

lpDialogFunc

[in] Pointer to the dialog box procedure. For more information about the dialog box procedure, see DialogProc.

dwInitParam

[in] Specifies the value to pass to the dialog box in the lParam parameter of the WM_INITDIALOG message.

Return Values

If the function succeeds, the return value is the value of the nResult parameter specified in the call to the EndDialog function used to terminate the dialog box.

If the function fails because the hWndParent parameter is invalid, the return value is zero. The function returns zero in this case for compatibility with previous versions of Windows. If the function fails for any other reason, the return value is –1. To get extended error information, call GetLastError.

Remarks

The DialogBoxParam function uses the CreateWindowEx function to create the dialog box. DialogBoxParam then sends a WM_INITDIALOG message (and a WM_SETFONT message if the template specifies the DS_SETFONT or DS_SHELLFONT style) to the dialog box procedure. The function displays the dialog box (regardless of whether the template specifies the WS_VISIBLE style), disables the owner window, and starts its own message loop to retrieve and dispatch messages for the dialog box.

When the dialog box procedure calls the EndDialog function, DialogBoxParam destroys the dialog box, ends the message loop, enables the owner window (if previously enabled), and returns the nResult parameter specified by the dialog box procedure when it called EndDialog.

Windows 95/98: The system can support a maximum of 255 controls per dialog box template. To place more than 255 controls in a dialog box, create the controls in the WM_INITDIALOG message handler rather than placing them in the template.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.104 DispatchMessage

The DispatchMessage function dispatches a message to a window procedure. It is typically used to dispatch a message retrieved by the GetMessage function.

```
DispatchMessage: procedure
(
    var lpmsg       :MSG
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__DispatchMessageA@4" );
```

Parameters

lpmsg

[in] Pointer to an MSG structure that contains the message.

Return Values

The return value specifies the value returned by the window procedure. Although its meaning depends on the message being dispatched, the return value generally is ignored.

Remarks

The MSG structure must contain valid message values. If the lpmsg parameter points to a WM_TIMER message and the lParam parameter of the WM_TIMER message is not NULL, lParam points to a function that is called instead of the window procedure.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.105 DlgDirList

The DlgDirList function replaces the contents of a list box with the names of the subdirectories and files in a specified directory. You can filter the list of names by specifying a set of file attributes. The list can optionally include mapped drives.

```
DlgDirList: procedure
(
    hDlg            :dword;
    lpPathSpec      :string;
    nIDListBox      :dword;
    nIDStackPath    :dword;
    uFileType       :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__DlgDirListA@20" );
```

Parameters

hDlg

[in] Handle to the dialog box that contains the list box.

lpPathSpec

[in/out] Pointer to a buffer containing a null-terminated string that specifies an absolute path, relative path, or filename. An absolute path can begin with a drive letter (for example, d:\) or a UNC name (for example, \\machinename\sharename).

The function splits the string into a directory and a filename. The function searches the directory for names that match the filename. If the string does not specify a directory, the function searches the current directory.

If the string includes a filename, the filename must contain at least one wildcard character (? or *). If the string does not include a filename, the function behaves as if you had specified the asterisk wildcard character (*) as the filename. All names in the specified directory that match the filename and have the attributes specified by the uFileType parameter are added to the list box.

nIDListBox

[in] Specifies the identifier of a list box in the hDlg dialog box. If this parameter is zero, DlgDirList does not try to fill a list box.

nIDStaticPath

[in] Specifies the identifier of a static control in the hDlg dialog box. DlgDirList sets the text of this control to display the current drive and directory. This parameter can be zero if you do not want to display the current drive and directory.

uFileType

[in] Specifies the attributes of the files or directories to be added to the list box. This parameter can be one or more of the following values.

Value	Description
DDL_ARCHIVE	Includes archived files.
DDL_DIRECTORY	Includes subdirectories. Subdirectory names are enclosed in square brackets ([ ]).
DDL_DRIVES	All mapped drives are added to the list. Drives are listed in the form [-x-], where x is the drive letter.
DDL_EXCLUSIVE	Includes only files with the specified attributes. By default, read-write files are listed even if DDL_READWRITE is not specified.
DDL_HIDDEN	Includes hidden files.
DDL_READONLY	Includes read-only files.
DDL_READWRITE	Includes read-write files with no additional attributes. This is the default setting.
DDL_SYSTEM	Includes system files.
DDL_POSTMSGS	If set, uses the PostMessage function to send messages to the list box. If not set, uses the SendMessage function.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. For example, if the string specified by lpPathSpec is not a valid path, the function fails. To get extended error information, call GetLastError.

Remarks

If lpPathSpec specifies a directory, DlgDirListComboBox changes the current directory to the specified directory before filling the list box. The text of the static control identified by the nIDStaticPath parameter is set to the name of the new current directory.

DlgDirList sends the LB_RESETCONTENT and LB_DIR messages to the list box.

Windows NT 4.0 and later: If uFileType includes the DDL_DIRECTORY flag and lpPathSpec specifies a first-level directory, such as C:\TEMP, the list box will always include a ".." entry for the root directory. This is true even if the root directory has hidden or system attributes and the DDL_HIDDEN and DDL_SYSTEM flags are not specified. The root directory of an NTFS volume has hidden and system attributes.

Windows NT/2000: The directory listing displays long filenames, if any.

Windows 95: The directory listing displays short filenames (the 8.3 form). You can use the SHGetFileInfo or Get-FullPathName functions to get the corresponding long filename.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.106 DlgDirListComboBox

The DlgDirListComboBox function replaces the contents of a combo box with the names of the subdirectories and files in a specified directory. You can filter the list of names by specifying a set of file attributes. The list of names can include mapped drive letters.

```
DlgDirListComboBox: procedure
(
    hDlg            :dword;
    lpPathSpec      :string;
    nIDComboBox     :dword;
    nIDStaticPath   :dword;
```

```
    uFiletype       :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__DlgDirListComboBoxA@20" );
```

Parameters

hDlg

[in] Handle to the dialog box that contains the combo box.

lpPathSpec

[in/out] Pointer to a buffer containing a null-terminated string that specifies an absolute path, relative path, or file name. An absolute path can begin with a drive letter (for example, d:\) or a UNC name (for example, \\machinename\sharename).

The function splits the string into a directory and a file name. The function searches the directory for names that match the file name. If the string does not specify a directory, the function searches the current directory.

If the string includes a file name, the file name must contain at least one wildcard character (? or *). If the string does not include a file name, the function behaves as if you had specified the asterisk wildcard character (*) as the file name. All names in the specified directory that match the file name and have the attributes specified by the uFileType parameter are added to the list displayed in the combo box.

nIDComboBox

[in] Specifies the identifier of a combo box in the hDlg dialog box. If this parameter is zero, DlgDirListComboBox does not try to fill a combo box.

nIDStaticPath

[in] Specifies the identifier of a static control in the hDlg dialog box. DlgDirListComboBox sets the text of this control to display the current drive and directory. This parameter can be zero if you don't want to display the current drive and directory.

uFiletype

[in] A set of bit flags that specify the attributes of the files or directories to be added to the combo box. This parameter can be a combination of the following values.

Value	Meaning
DDL_ARCHIVE	Includes archived files.
DDL_DIRECTORY	Includes subdirectories, which are enclosed in square brackets ([ ]).
DDL_DRIVES	All mapped drives are added to the list. Drives are listed in the form [-x-], where x is the drive letter.
DDL_EXCLUSIVE	Includes only files with the specified attributes. By default, read-write files are listed even if DDL_READWRITE is not specified.
DDL_HIDDEN	Includes hidden files.
DDL_READONLY	Includes read-only files.
DDL_READWRITE	Includes read-write files with no additional attributes. This is the default setting.
DDL_SYSTEM	Includes system files.
DDL_POSTMSGS	If this flag is set, uses the PostMessage function to send messages to the combo box. If this flag is not set, uses the SendMessage function.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. For example, if the string specified by lpPathSpec is not a valid path, the function fails. To get extended error information, call GetLastError.

Remarks

If lpPathSpec specifies a directory, DlgDirListComboBox changes the current directory to the specified directory before filling the combo box. The text of the static control identified by the nIDStaticPath parameter is set to the name of the new current directory.

DlgDirListComboBox sends the CB_RESETCONTENT and CB_DIR messages to the combo box.

Windows NT 4.0 and later: If uFileType includes the DDL_DIRECTORY flag and lpPathSpec specifies a first-level directory, such as C:\TEMP, the combo box will always include a ".." entry for the root directory. This is true even if the root directory has hidden or system attributes and the DDL_HIDDEN and DDL_SYSTEM flags are not specified. The root directory of an NTFS volume has hidden and system attributes.

Windows NT/ 2000: The list displays long file names, if any.

Windows 95: The list displays short file names (the 8.3 form). You can use the SHGetFileInfo or GetFullPath-Name functions to get the corresponding long file name.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.107 DlgDirSelectComboBoxEx

The DlgDirSelectComboBoxEx function retrieves the current selection from a combo box filled by using the DlgDirListComboBox function. The selection is interpreted as a drive letter, a file, or a directory name.

```
DlgDirSelectComboBoxEx: procedure
(
    hDlg            :dword;
    lpString        :string;
    nCount          :dword;
    nIDComboBox     :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__DlgDirSelectComboBoxExA@16" );
```

Parameters

hDlg

[in] Handle to the dialog box that contains the combo box.

lpString

[out] Pointer to the buffer that receives the selected path.

nCount

[in] Specifies the length, in characters, of the buffer pointed to by the lpString parameter.

nIDComboBox

[in] Specifies the integer identifier of the combo box control in the dialog box.

Return Values

If the current selection is a directory name, the return value is nonzero.

If the current selection is not a directory name, the return value is zero. To get extended error information, call GetLastError.

Remarks

If the current selection specifies a directory name or drive letter, the DlgDirSelectComboBoxEx function removes the enclosing square brackets (and hyphens for drive letters) so the name or letter is ready to be inserted into a new path or file name. If there is no selection, the contents of the buffer pointed to by lpString do not change.

The DlgDirSelectComboBoxEx function does not allow more than one file name to be returned from a combo box.

DlgDirSelectComboBoxEx sends CB_GETCURSEL and CB_GETLBTEXT messages to the combo box.

In the Win32 API, you can use this function with all three types of combo boxes (CBS_SIMPLE, CBS_DROPDOWN, and CBS_DROPDOWNLIST).

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.108 DlgDirSelectEx

The DlgDirSelectEx function retrieves the current selection from a single-selection list box. It assumes that the list box has been filled by the DlgDirList function and that the selection is a drive letter, filename, or directory name.

```
DlgDirSelectEx: procedure
(
    hDlg            :dword;
    lpString        :string;
    nCount          :dword;
    nIDListBox      :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__DlgDirSelectExA@16" );
```

Parameters

hDlg

[in] Handle to the dialog box that contains the list box.

lpString

[out] Pointer to a buffer that receives the selected path.

nCount

[in] Specifies the length, in TCHARs, of the buffer pointed to by lpString.

nIDListBox

[in] Specifies the identifier of a list box in the dialog box.

Return Values

If the current selection is a directory name, the return value is nonzero.

If the current selection is not a directory name, the return value is zero. To get extended error information, call GetLastError.

Remarks

The DlgDirSelectEx function copies the selection to the buffer pointed to by the lpString parameter. If the current selection is a directory name or drive letter, DlgDirSelectEx removes the enclosing square brackets (and hyphens, for drive letters), so that the name or letter is ready to be inserted into a new path. If there is no selection, lpString does not change.

DlgDirSelectEx sends LB_GETCURSEL and LB_GETTEXT messages to the list box. The function does not allow more than one filename to be returned from a list box. The list box must not be a multiple-selection list box. If it is, this function does not return a zero value and lpString remains unchanged.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.109 DragDetect

The DragDetect function captures the mouse and tracks its movement until the user releases the left button, presses the ESC key, or moves the mouse outside the drag rectangle around the specified point. The width and height of the drag rectangle are specified by the SM_CXDRAG and SM_CYDRAG values returned by the GetSystemMetrics function.

```
DragDetect: procedure
(
    hwnd        :dword;
    pt          :POINT
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__DragDetect@12" );
```

Parameters

hwnd

[in] Handle to the window receiving mouse input.

pt

[in] Initial position of the mouse, in screen coordinates. The function determines the coordinates of the drag rectangle by using this point.

Return Values

If the user moved the mouse outside of the drag rectangle while holding down the left button , the return value is nonzero.

If the user did not move the mouse outside of the drag rectangle while holding down the left button , the return value is zero.

Remarks

The system metrics for the drag rectangle are configurable, allowing for larger or smaller drag rectangles.

Requirements

Windows NT/2000: Requires Windows NT 4.0 or later.

Windows 95/98: Requires Windows 95 or later.

3.110 DrawAnimatedRects

The DrawAnimatedRects function draws a wire-frame rectangle and animates it to indicate the opening of an icon or the minimizing or maximizing of a window.

```
DrawAnimatedRects: procedure
(
        hwnd        :dword;
        idAni       :dword;
    var lprcFrom    :RECT;
    var lprcTo      :RECT
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__DrawAnimatedRects@16" );
```

Parameters

hwnd

[in] Handle to the window to which the rectangle is clipped. If this parameter is NULL, the working area of the screen is used.

idAni

[in] Specifies the type of animation. If you specify IDANI_CAPTION, the window caption will animate from the position specified by lprcFrom to the position specified by lprcTo. The effect is similar to minimizing or maximizing a window.

lprcFrom

[in] Pointer to a RECT structure specifying the location and size of the icon or minimized window. Coordinates are relative to the rectangle specified by the lprcClip parameter.

lprcTo

[in] Pointer to a RECT structure specifying the location and size of the restored window. Coordinates are relative to the rectangle specified by the lprcClip parameter.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Windows NT/ 2000: To get extended error information, call GetLastError.

Requirements

Windows NT/2000: Requires Windows NT 4.0 or later.

Windows 95/98: Requires Windows 95 or later.

3.111 DrawCaption

The DrawCaption function draws a window caption.

```
DrawCaption: procedure
(
        hwnd        :dword;
        hdc         :dword;
    var lprc        :RECT;
        uFlags      :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__DrawCaption@16" );
```

Parameters

hwnd

[in] Handle to a window that supplies text and an icon for the window caption.

hdc

[in] Handle to a device context. The function draws the window caption into this device context.

lprc

[in] Pointer to a RECT structure that specifies the bounding rectangle for the window caption.

uFlags

[in] Specifies drawing options. This parameter can be zero or more of the following values.

Value	Meaning
DC_ACTIVE	The function uses the colors that denote an active caption.
DC_GRADIENT	Windows 98, Windows 2000: When this flag is set, the function uses COLOR_GRADIENTACTIVECAPTION (if the DC_ACTIVE flag was set) or COLOR_GRADIENTINACTIVECAPTION for the title-bar color. If this flag is not set, the function uses COLOR_ACTIVECAPTION or COLOR_INACTIVECAPTION for both colors.
DC_ICON	The function draws the icon when drawing the caption text.
DC_INBUTTON	The function draws the caption as a button.
DC_SMALLCAP	The function draws a small caption, using the current small caption font.
DC_TEXT	The function draws the caption text when drawing the caption.

If DC_SMALLCAP is specified, the function draws a normal window caption.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Windows NT/ 2000: To get extended error information, call GetLastError.

Requirements

Windows NT/2000: Requires Windows NT 4.0 or later.

Windows 95/98: Requires Windows 95 or later.

3.112 DrawEdge

The DrawEdge function draws one or more edges of rectangle.

```
DrawEdge: procedure
(
        hdc         :dword;
    var qrc         :RECT;
        edge        :dword;
        grfFlags    :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__DrawEdge@16" );
```

Parameters

hdc

[in] Handle to the device context.

qrc

[in/out] Pointer to a RECT structure that contains the logical coordinates of the rectangle.

edge

[in] Specifies the type of inner and outer edges to draw. This parameter must be a combination of one inner-border flag and one outer-border flag. The inner-border flags are as follows.

Value	Meaning
BDR_RAISEDINNER	Raised inner edge.
BDR_SUNKENINNER	Sunken inner edge.

The outer-border flags are as follows.

Value	Meaning
BDR_RAISEDOUTER	Raised outer edge.
BDR_SUNKENOUTER	Sunken outer edge.

Alternatively, the edge parameter can specify one of the following flags.

Value	Meaning
EDGE_BUMP	Combination of BDR_RAISEDOUTER and BDR_SUNKENINNER.
EDGE_ETCHED	Combination of BDR_SUNKENOUTER and BDR_RAISEDINNER.
EDGE_RAISED	Combination of BDR_RAISEDOUTER and BDR_RAISEDINNER.
EDGE_SUNKEN	Combination of BDR_SUNKENOUTER and BDR_SUNKENINNER.

grfFlags

[in] Specifies the type of border. This parameter can be a combination of the following values.

Value	Meaning

BF_ADJUST	If this flag is passed, shrink the rectangle pointed to by the qrc parameter to exclude the edges that were drawn. If this flag is not passed, then do not change the rectangle pointed to by the qrc parameter. .
BF_BOTTOM	Bottom of border rectangle.
BF_BOTTOMLEFT	Bottom and left side of border rectangle.
BF_BOTTOMRIGHT	Bottom and right side of border rectangle.
BF_DIAGONAL	Diagonal border.
BF_DIAGONAL_ENDBOTTOMLEFT	Diagonal border. The end point is the bottom-left corner of the rectangle; the origin is top-right corner.
BF_DIAGONAL_ENDBOTTOMRIGHT	Diagonal border. The end point is the bottom-right corner of the rectangle; the origin is top-left corner.
BF_DIAGONAL_ENDTOPLEFT	Diagonal border. The end point is the top-left corner of the rectangle; the origin is bottom-right corner.
BF_DIAGONAL_ENDTOPRIGHT	Diagonal border. The end point is the top-right corner of the rectangle; the origin is bottom-left corner.
BF_FLAT	Flat border.
BF_LEFT	Left side of border rectangle.
BF_MIDDLE	Interior of rectangle to be filled.
BF_MONO	One-dimensional border.
BF_RECT	Entire border rectangle.
BF_RIGHT	Right side of border rectangle.
BF_SOFT	Soft buttons instead of tiles.
BF_TOP	Top of border rectangle.
BF_TOPLEFT	Top and left side of border rectangle.
BF_TOPRIGHT	Top and right side of border rectangle.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Windows NT/ 2000: To get extended error information, call GetLastError.

Requirements

Windows NT/2000: Requires Windows NT 3.51 or later.

Windows 95/98: Requires Windows 95 or later.

3.113 DrawFocusRect

The DrawFocusRect function draws a rectangle in the style used to indicate that the rectangle has the focus.

```
DrawFocusRect: procedure
(
        hDC         :dword;
    var lprc        :RECT
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__DrawFocusRect@8" );
```

Parameters

hDC

[in] Handle to the device context.

lprc

[in] Pointer to a RECT structure that specifies the logical coordinates of the rectangle.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Windows NT/ 2000: To get extended error information, call GetLastError.

Remarks

DrawFocusRect works only in MM_TEXT mode.

Because DrawFocusRect is an XOR function, calling it a second time with the same rectangle removes the rectangle from the screen.

This function draws a rectangle that cannot be scrolled. To scroll an area containing a rectangle drawn by this function, call DrawFocusRect to remove the rectangle from the screen, scroll the area, and then call DrawFocusRect again to draw the rectangle in the new position.

Whistler: The focus rectangle can now be thicker than 1 pixel, so it is more visible for high-resolution, high-density displays and accessibility needs. This is handled by the SPI_SETFOCUSBORDERWIDTH and SPI_SETFOCUSBORDERHEIGHT in SystemParametersInfo.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.114 DrawFrameControl

The DrawFrameControl function draws a frame control of the specified type and style.

```
DrawFrameControl: procedure
(
        hdc     :dword;
    var lprc    :RECT;
        uType   :dword;
        uState  :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__DrawFrameControl@16" );
```

Parameters

hdc

[in] Handle to the device context of the window in which to draw the control.

lprc

[in] Pointer to a RECT structure that contains the logical coordinates of the bounding rectangle for frame control.

uType

[in] Specifies the type of frame control to draw. This parameter can be one of the following values.

Value	Meaning

DFC_BUTTON	Standard button
DFC_CAPTION	Title bar
DCF_MENU	Menu bar
DFC_POPUPMENU	Windows 98, Windows 2000: Popup menu item.
DFC_SCROLL	Scroll bar

uState

[in] Specifies the initial state of the frame control. If uType is DFC_BUTTON, uState can be one of the following values.

Value	Meaning
DFCS_BUTTON3STATE	Three-state button
DFCS_BUTTONCHECK	Check box
DFCS_BUTTONPUSH	Push button
DFCS_BUTTONRADIO	Radio button
DFCS_BUTTONRADIOIMAGE	Image for radio button (nonsquare needs image)
DFCS_BUTTONRADIOMASK	Mask for radio button (nonsquare needs mask)

If uType is DFC_CAPTION, uState can be one of the following values.

Value	Meaning
DFCS_CAPTIONCLOSE	Close button
DFCS_CAPTIONHELP	Help button
DFCS_CAPTIONMAX	Maximize button
DFCS_CAPTIONMIN	Minimize button
DFCS_CAPTIONRESTORE	Restore button

If uType is DFC_MENU, uState can be one of the following values.

Value	Meaning
DFCS_MENUARROW	Submenu arrow
DFCS_MENUARROWRIGHT	Submenu arrow pointing left. This is used for the right-to-left cascading menus used with right-to-left languages such as Arabic or Hebrew.
DFCS_MENUBULLET	Bullet
DFCS_MENUCHECK	Check mark

If uType is DFC_SCROLL, uState can be one of the following values.

Value	Meaning
DFCS_SCROLLCOMBOBOX	Combo box scroll bar
DFCS_SCROLLDOWN	Down arrow of scroll bar
DFCS_SCROLLLEFT	Left arrow of scroll bar
DFCS_SCROLLRIGHT	Right arrow of scroll bar
DFCS_SCROLLSIZEGRIP	Size grip in bottom-right corner of window
DFCS_SCROLLSIZEGRIPRIGHT	Size grip in bottom-left corner of window. This is used with right-to-left languages such as Arabic or Hebrew.
DFCS_SCROLLUP	Up arrow of scroll bar

The following style can be used to adjust the bounding rectangle of the push button.

Value	Meaning
DFCS_ADJUSTRECT	Bounding rectangle is adjusted to exclude the surrounding edge of the push button.

One or more of the following values can be used to set the state of the control to be drawn.

Value	Meaning
DFCS_CHECKED	Button is checked.
DFCS_FLAT	Button has a flat border.
DFCS_HOT	Windows 98, Windows 2000: Button is hot-tracked.
DFCS_INACTIVE	Button is inactive (grayed).
DFCS_MONO	Button has a monochrome border.
DFCS_PUSHED	Button is pushed.

DFCS_TRANSPARENT	Windows 98, Windows 2000: The background remains untouched.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Windows NT/ 2000: To get extended error information, call GetLastError.

Remarks

If uType is either DFC_MENU or DFC_BUTTON and uState is not DFCS_BUTTONPUSH, the frame control is a black-on-white mask (that is, a black frame control on a white background). In such cases, the application must pass a handle to a bitmap memory device control. The application can then use the associated bitmap as the hbm-Mask parameter to the MaskBlt function, or it can use the device context as a parameter to the BitBlt function using ROPs such as SRCAND and SRCINVERT.

Requirements

Windows NT/2000: Requires Windows NT 3.51 or later.

Windows 95/98: Requires Windows 95 or later.

3.115 DrawIcon

The DrawIcon function draws an icon or cursor into the specified device context.

To specify additional drawing options, use the DrawIconEx function.

```
DrawIcon: procedure
(
    hDC         :dword;
    X           :dword;
    Y           :dword;
    hIcon       :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__DrawIcon@16" );
```

Parameters

hDC

[in] Handle to the device context into which the icon or cursor will be drawn.

X

[in] Specifies the logical x-coordinate of the upper-left corner of the icon.

Y

[in] Specifies the logical y-coordinate of the upper-left corner of the icon.

hIcon

[in] Handle to the icon to be drawn.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

DrawIcon places the icon's upper-left corner at the location specified by the X and Y parameters. The location is subject to the current mapping mode of the device context.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.116 DrawIconEx

The DrawIconEx function draws an icon or cursor into the specified device context, performing the specified raster operations, and stretching or compressing the icon or cursor as specified.

```
DrawIconEx: procedure
(
    hdc                 :dword;
    xLeft               :dword;
    yTop                :dword;
    hIcon               :dword;
    cxWidth             :dword;
    cyWidth             :dword;
    istepIfAniCur       :dword;
    hbrFlickerFreeDraw  :dword;
    diFlags             :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__DrawIconEx@36" );
```

Parameters

hdc

[in] Handle to the device context into which the icon or cursor will be drawn.

xLeft

[in] Specifies the logical x-coordinate of the upper-left corner of the icon or cursor.

yTop

[in] Specifies the logical y-coordinate of the upper-left corner of the icon or cursor.

hIcon

[in] Handle to the icon or cursor to be drawn. This parameter can identify an animated cursor.

cxWidth

[in] Specifies the logical width of the icon or cursor. If this parameter is zero and the diFlags parameter is DI_DEFAULTSIZE, the function uses the SM_CXICON or SM_CXCURSOR system metric value to set the width. If this parameter is zero and DI_DEFAULTSIZE is not used, the function uses the actual resource width.

cyWidth

[in] Specifies the logical height of the icon or cursor. If this parameter is zero and the diFlags parameter is DI_DEFAULTSIZE, the function uses the SM_CYICON or SM_CYCURSOR system metric value to set the

width. If this parameter is zero and DI_DEFAULTSIZE is not used, the function uses the actual resource height.

istepIfAniCur

[in] Specifies the index of the frame to draw, if hIcon identifies an animated cursor. This parameter is ignored if hIcon does not identify an animated cursor.

hbrFlickerFreeDraw

[in] Handle to a brush that the system uses for flicker-free drawing. If hbrFlickerFreeDraw is a valid brush handle, the system creates an offscreen bitmap using the specified brush for the background color, draws the icon or cursor into the bitmap, and then copies the bitmap into the device context identified by hdc. If hbrFlickerFreeDraw is NULL, the system draws the icon or cursor directly into the device context.

diFlags

[in] Specifies the drawing flags. This parameter can be one of the following values:

Value	Meaning
DI_COMPAT	Draws the icon or cursor using the system default image rather than the user-specified image.
DI_DEFAULTSIZE	Draws the icon or cursor using the width and height specified by the system metric values for cursors or icons, if the cxWidth and cyWidth parameters are set to zero. If this flag is not specified and cxWidth and cyWidth are set to zero, the function uses the actual resource size.
DI_IMAGE	Draws the icon or cursor using the image.
DI_MASK	Draws the icon or cursor using the mask.
DI_NORMAL	Combination of DI_IMAGE and DI_MASK.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

The DrawIconEx function places the icon's upper-left corner at the location specified by the xLeft and yTop parameters. The location is subject to the current mapping mode of the device context.

Requirements

Windows NT/2000: Requires Windows NT 3.5 or later.

Windows 95/98: Requires Windows 95 or later.

3.117 DrawMenuBar

The DrawMenuBar function redraws the menu bar of the specified window. If the menu bar changes after the system has created the window, this function must be called to draw the changed menu bar.

```
DrawMenuBar: procedure
(
    hWnd        :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__DrawMenuBar@4" );
```

Parameters

hWnd

[in] Handle to the window whose menu bar needs redrawing.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.118 DrawState

The DrawState function displays an image and applies a visual effect to indicate a state, such as a disabled or default state.

```
DrawState: procedure
(
    hdc             :dword;
    hbr             :dword;
    lpOutputFunc    :DRAWSTATEPROC;
    lData           :dword;
    wData           :dword;
    x               :dword;
    y               :dword;
    cx              :dword;
    cy              :dword;
    fuFlags         :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__DrawStateA@40" );
```

Parameters

hdc

[in] Handle to the device context to draw in.

hbr

[in] Handle to the brush used to draw the image, if the state specified by the fuFlags parameter is DSS_MONO. This parameter is ignored for other states.

lpOutputFunc

[in] Pointer to an application-defined callback function used to render the image. This parameter is required if the image type in fuFlags is DST_COMPLEX. It is optional and can be NULL if the image type is DST_TEXT. For all other image types, this parameter is ignored. For more information about the callback function, see the DrawStateProc function.

lData

[in] Specifies information about the image. The meaning of this parameter depends on the image type.

wData

[in] Specifies information about the image. The meaning of this parameter depends on the image type. It is, however, zero extended for use with the DrawStateProc function.

x

[in] Specifies the horizontal location at which to draw the image.

y

[in] Specifies the vertical location at which to draw the image.

cx

[in] Specifies the width of the image, in device units. This parameter is required if the image type is DST_COMPLEX. Otherwise, it can be zero to calculate the width of the image.

cy

[in] Specifies the height of the image, in device units. This parameter is required if the image type is DST_COMPLEX. Otherwise, it can be zero to calculate the height of the image.

fuFlags

[in] Specifies the image type and state. This parameter can be one of the following type values.

Value (type)	Meaning
DST_BITMAP	The image is a bitmap. The lData parameter is the bitmap handle. Note that the bitmap cannot already be selected into an existing device context.
DST_COMPLEX	The image is application defined. To render the image, calls the callback function specified by the lpOutputFunc parameter.
DST_ICON	The image is an icon. The lData parameter is the icon handle.
DST_PREFIXTEXT	The image is text that may contain an accelerator mnemonic. interprets the ampersand (&) prefix character as a directive to underscore the character that follows. The lData parameter is a pointer to the string, and the wData parameter specifies the length. If wData is zero, the string must be null-terminated.
DST_TEXT	The image is text. The lData parameter is a pointer to the string, and the wData parameter specifies the length. If wData is zero, the string must be null-terminated.

This parameter can also be one of the following state values.

Value (state)	Meaning
DSS_DISABLED	Embosses the image.
DSS_HIDEPREFIX	Windows 2000: Ignores the ampersand (&) prefix character in the text, thus the letter that follows will not be underlined. This must be used with DST_PREFIXTEXT.
DSS_MONO	Draws the image using the brush specified by the hbr parameter.
DSS_NORMAL	Draws the image without any modification.
DSS_PREFIXONLY	Windows 2000: Draws only the underline at the position of the letter after the ampersand (&) prefix character. No text in the string is drawn. This must be used with DST_PREFIXTEXT.
DSS_RIGHT	Aligns the text to the right.
DSS_UNION	Dithers the image.

For all states except DSS_NORMAL, the image is converted to monochrome before the visual effect is applied.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Windows NT/ 2000: To get extended error information, call GetLastError.

Requirements

Windows NT/2000: Requires Windows NT 4.0 or later.

Windows 95/98: Requires Windows 95 or later.

3.119 DrawText

The DrawText function draws formatted text in the specified rectangle. It formats the text according to the specified method (expanding tabs, justifying characters, breaking lines, and so forth).

To specify additional formatting options, use the DrawTextEx function.

```
DrawText: procedure
(
        hDC         :dword;
        lpString    :string;
        nCount      :dword;
    var lpRect      :RECT;
        uFormat     :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__DrawTextA@20" );
```

Parameters

hDC

[in] Handle to the device context.

lpString

[in] Pointer to the string that specifies the text to be drawn. If the nCount parameter is –1, the string must be null-terminated.

If uFormat includes DT_MODIFYSTRING, the function could add up to four additional characters to this string. The buffer containing the string should be large enough to accommodate these extra characters.

nCount

[in] Specifies the length of the string. For the ANSI function it is a BYTE count and for the Unicode function it is a WORD count. Note that for the ANSI function, characters in SBCS code pages take one byte each, while most characters in DBCS code pages take two bytes; for the Unicode function, most currently defined Unicode characters (those in the Basic Multilingual Plane (BMP)) are one WORD while Unicode surrogates are two WORDs. If nCount is –1, then the lpString parameter is assumed to be a pointer to a null-terminated string and DrawText computes the character count automatically.

Windows 95/98: This number many not exceed 8192.

lpRect

[in/out] Pointer to a RECT structure that contains the rectangle (in logical coordinates) in which the text is to be formatted.

uFormat

[in] Specifies the method of formatting the text. This parameter can be one or more of the following values.

Value	Description
DT_BOTTOM	Justifies the text to the bottom of the rectangle. This value is used only with the DT_SINGLELINE value.
DT_CALCRECT	Determines the width and height of the rectangle. If there are multiple lines of text, uses the width of the rectangle pointed to by the lpRect parameter and extends the base of the rectangle to bound the last line of text. If there is only one line of text, modifies the right side of the rectangle so that it bounds the last character in the line. In either case, returns the height of the formatted text but does not draw the text.
DT_CENTER	Centers text horizontally in the rectangle.
DT_EDITCONTROL	Duplicates the text-displaying characteristics of a multiline edit control. Specifically, the average character width is calculated in the same manner as for an edit control, and the function does not display a partially visible last line.
DT_END_ELLIPSIS	For displayed text, if the end of a string does not fit in the rectangle, it is truncated and ellipses are added. If a word that is not at the end of the string goes beyond the limits of the rectangle, it is truncated without ellipses. The string is not modified unless the DT_MODIFYSTRING flag is specified. Compare with DT_PATH_ELLIPSIS and DT_WORD_ELLIPSIS.
DT_EXPANDTABS	Expands tab characters. The default number of characters per tab is eight. The DT_WORD_ELLIPSIS, DT_PATH_ELLIPSIS, and DT_END_ELLIPSIS values cannot be used with the DT_EXPANDTABS value.
DT_EXTERNALLEADING	Includes the font external leading in line height. Normally, external leading is not included in the height of a line of text.

DT_HIDEPREFIX	Windows 2000: Ignores the ampersand (&) prefix character in the text. The letter that follows will not be underlined, but other mnemonic-prefix characters are still processed. For example:
	
	`input string:  "A&bc&&d"`
	`normal:  "Abc&d"`
	`DT_HIDEPREFIX:  "Abc&d"`
	
	Compare with DT_NOPREFIX and DT_PREFIXONLY.

DT_INTERNAL	Uses the system font to calculate text metrics.
DT_LEFT	Aligns text to the left.
DT_MODIFYSTRING	Modifies the specified string to match the displayed text. This value has no effect unless DT_END_ELLIPSIS or DT_PATH_ELLIPSIS is specified.
DT_NOCLIP	Draws without clipping. is somewhat faster when DT_NOCLIP is used.
DT_NOFULLWIDTHCHARBREAK	Windows 98, Windows 2000: Prevents a line break at a DBCS (double-wide character string), so that the line breaking rule is equivalent to SBCS strings. For example, this can be used in Korean windows, for more readability of icon labels. This value has no effect unless DT_WORDBREAK is specified.

DT_NOPREFIX — Turns off processing of prefix characters. Normally, interprets the mnemonic-prefix character & as a directive to underscore the character that follows, and the mnemonic-prefix characters && as a directive to print a single &. By specifying DT_NOPREFIX, this processing is turned off. For example,

```
input string:   "A&bc&&d"

normal:         "Abc&d"

DT_NOPREFIX:    "A&bc&&d"
```

Compare with DT_HIDEPREFIX and DT_PREFIXONLY.

DT_PATH_ELLIPSIS — For displayed text, replaces characters in the middle of the string with ellipses so that the result fits in the specified rectangle. If the string contains backslash (\) characters, DT_PATH_ELLIPSIS preserves as much as possible of the text after the last backslash.

The string is not modified unless the DT_MODIFYSTRING flag is specified.

Compare with DT_END_ELLIPSIS and DT_WORD_ELLIPSIS.

DT_PREFIXONLY — Windows 2000: Draws only an underline at the position of the character following the ampersand (&) prefix character. Does not draw any other characters in the string. For example,

```
input string:    "A&bc&&d"

normal:          "Abc&d"

DT_PREFIXONLY:   " _    "
```

Compare with DT_HIDEPREFIX and DT_NOPREFIX.

DT_RIGHT — Aligns text to the right.

DT_RTLREADING — Layout in right-to-left reading order for bi-directional text when the font selected into the hdc is a Hebrew or Arabic font. The default reading order for all text is left-to-right.

DT_SINGLELINE — Displays text on a single line only. Carriage returns and line feeds do not break the line.

DT_TABSTOP — Sets tab stops. Bits 15–8 (high-order byte of the low-order word) of the uFormat parameter specify the number of characters for each tab. The default number of characters per tab is eight.The DT_CALCRECT, DT_EXTERNALLEADING, DT_INTERNAL, DT_NOCLIP, and DT_NOPREFIX values cannot be used with the DT_TABSTOP value.

DT_TOP — Justifies the text to the top of the rectangle.

DT_VCENTER — Centers text vertically. This value is used only with the DT_SINGLELINE value.

DT_WORDBREAK — Breaks words. Lines are automatically broken between words if a word would extend past the edge of the rectangle specified by the lpRect parameter. A carriage return-line feed sequence also breaks the line.

DT_WORD_ELLIPSIS — Truncates any word that does not fit in the rectangle and adds ellipses.

Compare with DT_END_ELLIPSIS and DT_PATH_ELLIPSIS.

Return Values

If the function succeeds, the return value is the height of the text in logical units. If DT_VCENTER or DT_BOTTOM is specified, the return value is the offset from lpRect->top to the bottom of the drawn text

If the function fails, the return value is zero.

Windows NT/ 2000: To get extended error information, call GetLastError.

Remarks

The DrawText function uses the device context's selected font, text color, and background color to draw the text. Unless the DT_NOCLIP format is used, DrawText clips the text so that it does not appear outside the specified rectangle. All formatting is assumed to have multiple lines unless the DT_SINGLELINE format is specified.

If the selected font is too large for the specified rectangle, the DrawText function does not attempt to substitute a smaller font.

The DrawText function supports only fonts whose escapement and orientation are both zero.

The text alignment mode for the device context must include the TA_LEFT, TA_TOP, and TA_NOUPDATECP flags.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.120 DrawTextEx

The DrawTextEx function draws formatted text in the specified rectangle.

```
DrawTextEx: procedure
(
        hdc         :dword;
        lpchText    :string;
        cchText     :dword;
    var lprc        :RECT;
        dwDTFormat  :dword;
    var lpDTParams  :DRAWTEXTPARAMS
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__DrawTextExA@24" );
```

Parameters

hdc

[in] Handle to the device context in which to draw.

lpchText

[in/out] Pointer to the string that contains the text to draw. The string must be null-terminated if the cchText parameter is –1.

If dwDTFormat includes DT_MODIFYSTRING, the function could add up to four additional characters to this string. The buffer containing the string should be large enough to accommodate these extra characters.

cchText

[in] Specifies the length of the string specified by the lpchText parameter. For the ANSI function it is a BYTE count and for the Unicode function it is a WORD count. Note that for the ANSI function, characters in SBCS

code pages take one byte each, while most characters in DBCS code pages take two bytes; for the Unicode function, most currently defined Unicode characters (those in the Basic Multilingual Plane (BMP)) are one WORD while Unicode surrogates are two WORDs. If the string is null-terminated, this parameter can be –1 to calculate the length.

Windows 95/98: This number may not exceed 8192.

lprc

[in/out] Pointer to a RECT structure that contains the rectangle, in logical coordinates, in which the text is to be formatted.

dwDTFormat

[in] Specifies formatting options. This parameter can be one or more of the following values.

Value	Meaning
DT_BOTTOM	Justifies the text to the bottom of the rectangle. This value is used only with the DT_SINGLELINE value.
DT_CALCRECT	Determines the width and height of the rectangle. If there are multiple lines of text, uses the width of the rectangle pointed to by the lprc parameter and extends the base of the rectangle to bound the last line of text. If there is only one line of text, modifies the right side of the rectangle so that it bounds the last character in the line. In either case, returns the height of the formatted text, but does not draw the text.
DT_CENTER	Centers text horizontally in the rectangle.
DT_EDITCONTROL	Duplicates the text-displaying characteristics of a multiline edit control. Specifically, the average character width is calculated in the same manner as for an edit control, and the function does not display a partially visible last line.
DT_END_ELLIPSIS	For displayed text, replaces the end of a string with ellipses so that the result fits in the specified rectangle. Any word (not at the end of the string) that goes beyond the limits of the rectangle is truncated without ellipses. The string is not modified unless the DT_MODIFYSTRING flag is specified. Compare with DT_PATH_ELLIPSIS and DT_WORD_ELLIPSIS.
DT_EXPANDTABS	Expands tab characters. The default number of characters per tab is eight.
DT_EXTERNALLEADING	Includes the font external leading in line height. Normally, external leading is not included in the height of a line of text.

DT_HIDEPREFIX	Windows 2000: Ignores the ampersand (&) prefix character in the text. The letter that follows will not be underlined, but other mnemonic-prefix characters are still processed. Example: input string: "A&bc&&d" normal: "Abc&d" HIDEPREFIX: "Abc&d" Compare with DT_NOPREFIX and DT_PREFIXONLY.
DT_INTERNAL	Uses the system font to calculate text metrics.
DT_LEFT	Aligns text to the left.
DT_MODIFYSTRING	Modifies the specified string to match the displayed text. This value has no effect unless DT_END_ELLIPSIS or DT_PATH_ELLIPSIS is specified.
DT_NOCLIP	Draws without clipping. is somewhat faster when DT_NOCLIP is used.
DT_NOFULLWIDTHCHARBREAK	Windows 98, Windows 2000: Prevents a line break at a DBCS (double-wide character string), so that the line-breaking rule is equivalent to SBCS strings. For example, this can be used in Korean windows, for more readability of icon labels. This value has no effect unless DT_WORDBREAK is specified.
DT_NOPREFIX	Turns off processing of prefix characters. Normally, interprets the ampersand (&) mnemonic-prefix character as a directive to underscore the character that follows, and the double-ampersand (&&) mnemonic-prefix characters as a directive to print a single ampersand. By specifying DT_NOPREFIX, this processing is turned off. Compare with DT_HIDEPREFIX and DT_PREFIXONLY
DT_PATH_ELLIPSIS	For displayed text, replaces characters in the middle of the string with ellipses so that the result fits in the specified rectangle. If the string contains backslash (\) characters, DT_PATH_ELLIPSIS preserves as much as possible of the text after the last backslash. The string is not modified unless the DT_MODIFYSTRING flag is specified. Compare with DT_END_ELLIPSIS and DT_WORD_ELLIPSIS.

DT_PREFIXONLY	Windows 2000: Draws only an underline at the position of the character following the ampersand (&) prefix character. Does not draw any character in the string. Example: input string: "A&bc&&d" normal: "Abc&d" PREFIXONLY: " " Compare with DT_NOPREFIX and DT_HIDEPREFIX.
DT_RIGHT	Aligns text to the right.
DT_RTLREADING	Layout in right-to-left reading order for bi-directional text when the font selected into the hdc is a Hebrew or Arabic font. The default reading order for all text is left-to-right.
DT_SINGLELINE	Displays text on a single line only. Carriage returns and line feeds do not break the line.
DT_TABSTOP	Sets tab stops. The DRAWTEXTPARAMS structure pointed to by the lpDTParams parameter specifies the number of average character widths per tab stop.
DT_TOP	Justifies the text to the top of the rectangle.
DT_VCENTER	Centers text vertically. This value is used only with the DT_SINGLELINE value.
DT_WORDBREAK	Breaks words. Lines are automatically broken between words if a word extends past the edge of the rectangle specified by the lprc parameter. A carriage return-line feed sequence also breaks the line.
DT_WORD_ELLIPSIS	Truncates any word that does not fit in the rectangle and adds ellipses. Compare with DT_END_ELLIPSIS and DT_PATH_ELLIPSIS.

lpDTParams

[in] Pointer to a DRAWTEXTPARAMS structure that specifies additional formatting options. This parameter can be NULL.

Return Values

If the function succeeds, the return value is the text height in logical units. If DT_VCENTER or DT_BOTTOM is specified, the return value is the offset from lprc->top to the bottom of the drawn text

If the function fails, the return value is zero.

Windows NT/ 2000: To get extended error information, call GetLastError.

Remarks

The DrawTextEx function supports only fonts whose escapement and orientation are both zero.

The text alignment mode for the device context must include the TA_LEFT, TA_TOP, and TA_NOUPDATECP flags.

Requirements

Windows NT/2000: Requires Windows NT 4.0 or later.

Windows 95/98: Requires Windows 95 or later.

3.121 EmptyClipboard

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

```
EmptyClipboard: procedure;
    @stdcall;
    @returns( "eax" );
    @external( "__imp__EmptyClipboard@0" );
```

Parameters

This function has no parameters.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

Before calling EmptyClipboard, an application must open the clipboard by using the OpenClipboard function. If the application specifies a NULL window handle when opening the clipboard, EmptyClipboard succeeds but sets the clipboard owner to NULL.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.122 EnableMenuItem

The EnableMenuItem function enables, disables, or grays the specified menu item.

```
EnableMenuItem: procedure
(
    hMenu           :dword;
    uIDEnableItem   :dword;
    uEnable         :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__EnableMenuItem@12" );
```

Parameters

hMenu

[in] Handle to the menu.

uIDEnableItem

[in] Specifies the menu item to be enabled, disabled, or grayed, as determined by the uEnable parameter. This parameter specifies an item in a menu bar, menu, or submenu.

uEnable

[in] Controls the interpretation of the uIDEnableItem parameter and indicate whether the menu item is enabled, disabled, or grayed. This parameter must be a combination of either MF_BYCOMMAND or MF_BYPOSITION and MF_ENABLED, MF_DISABLED, or MF_GRAYED.

Value	Meaning
MF_BYCOMMAND	Indicates that uIDEnableItem gives the identifier of the menu item. If neither the MF_BYCOMMAND nor MF_BYPOSITION flag is specified, the MF_BYCOMMAND flag is the default flag.
MF_BYPOSITION	Indicates that uIDEnableItem gives the zero-based relative position of the menu item.
MF_DISABLED	Indicates that the menu item is disabled, but not grayed, so it cannot be selected.
MF_ENABLED	Indicates that the menu item is enabled and restored from a grayed state so that it can be selected.
MF_GRAYED	Indicates that the menu item is disabled and grayed so that it cannot be selected.

Return Values

The return value specifies the previous state of the menu item (it is either MF_DISABLED, MF_ENABLED, or MF_GRAYED). If the menu item does not exist, the return value is -1.

Remarks

An application must use the MF_BYPOSITION flag to specify the correct menu handle. If the menu handle to the menu bar is specified, the top-level menu item (an item in the menu bar) is affected. To set the state of an item in a drop-down menu or submenu by position, an application must specify a handle to the drop-down menu or submenu.

When an application specifies the MF_BYCOMMAND flag, the system checks all items that open submenus in the menu identified by the specified menu handle. Therefore, unless duplicate menu items are present, specifying the menu handle to the menu bar is sufficient.

The InsertMenu, InsertMenuItem, LoadMenuIndirect, ModifyMenu, and SetMenuItemInfo functions can also set the state (enabled, disabled, or grayed) of a menu item.

When you change a window menu, the menu bar is not immediately updated. To force the update, call Draw-MenuBar.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.123 EnableScrollBar

The EnableScrollBar function enables or disables one or both scroll bar arrows.

```
EnableScrollBar: procedure
(
    hWnd            :dword;
    wSBflags        :dword;
    wArrows         :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__EnableScrollBar@12" );
```

Parameters

hWnd

[in] Handle to a window or a scroll bar control, depending on the value of the wSBflags parameter.

wSBflags

[in] Specifies the scroll bar type. This parameter can be one of the following values.

Value	Meaning
SB_BOTH	Enables or disables the arrows on the horizontal and vertical scroll bars associated with the specified window. The hWnd parameter must be the handle to the window.
SB_CTL	Indicates that the scroll bar is a scroll bar control. The hWnd must be the handle to the scroll bar control.
SB_HORZ	Enables or disables the arrows on the horizontal scroll bar associated with the specified window. The hWnd parameter must be the handle to the window.
SB_VERT	Enables or disables the arrows on the vertical scroll bar associated with the specified window. The hWnd parameter must be the handle to the window.

wArrows

[in] Specifies whether the scroll bar arrows are enabled or disabled and indicates which arrows are enabled or disabled. This parameter can be one of the following values.

Value	Meaning
ESB_DISABLE_BOTH	Disables both arrows on a scroll bar.
ESB_DISABLE_DOWN	Disables the down arrow on a vertical scroll bar.
ESB_DISABLE_LEFT	Disables the left arrow on a horizontal scroll bar.
ESB_DISABLE_LTUP	Disables the left arrow on a horizontal scroll bar or the up arrow of a vertical scroll bar.
ESB_DISABLE_RIGHT	Disables the right arrow on a horizontal scroll bar.
ESB_DISABLE_RTDN	Disables the right arrow on a horizontal scroll bar or the down arrow of a vertical scroll bar.
ESB_DISABLE_UP	Disables the up arrow on a vertical scroll bar.
ESB_ENABLE_BOTH	Enables both arrows on a scroll bar.

Return Values

If the arrows are enabled or disabled as specified, the return value is nonzero.

If the arrows are already in the requested state or an error occurs, the return value is zero. To get extended error information, call GetLastError.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.124 EnableWindow

The EnableWindow function enables or disables mouse and keyboard input to the specified window or control. When input is disabled, the window does not receive input such as mouse clicks and key presses. When input is enabled, the window receives all input.

```
EnableWindow: procedure
```

```
(
    hWnd            :dword;
    bEnable         :boolean
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__EnableWindow@8" );
```

Parameters

hWnd

[in] Handle to the window to be enabled or disabled.

bEnable

[in] Specifies whether to enable or disable the window. If this parameter is TRUE, the window is enabled. If the parameter is FALSE, the window is disabled.

Return Values

If the window was previously disabled, the return value is nonzero.

If the window was not previously disabled, the return value is zero. To get extended error information, call GetLastError.

Remarks

If the window is being disabled, the system sends a WM_CANCELMODE message. If the enabled state of a window is changing, the system sends a WM_ENABLE message after the WM_CANCELMODE message. (These messages are sent before EnableWindow returns.) If a window is already disabled, its child windows are implicitly disabled, although they are not sent a WM_ENABLE message.

A window must be enabled before it can be activated. For example, if an application is displaying a modeless dialog box and has disabled its main window, the application must enable the main window before destroying the dialog box. Otherwise, another window will receive the keyboard focus and be activated. If a child window is disabled, it is ignored when the system tries to determine which window should receive mouse messages.

By default, a window is enabled when it is created. To create a window that is initially disabled, an application can specify the WS_DISABLED style in the CreateWindow or CreateWindowEx function. After a window has been created, an application can use EnableWindow to enable or disable the window.

An application can use this function to enable or disable a control in a dialog box. A disabled control cannot receive the keyboard focus, nor can a user gain access to it.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.125 EndDeferWindowPos

The EndDeferWindowPos function simultaneously updates the position and size of one or more windows in a single screen-refreshing cycle.

```
EndDeferWindowPos: procedure
(
    hWinPosInfo     :dword
);
```

```
    @stdcall;
    @returns( "eax" );
    @external( "__imp__EndDeferWindowPos@4" );
```

Parameters

hWinPosInfo

[in] Handle to a multiple-window – position structure that contains size and position information for one or more windows. This internal structure is returned by the BeginDeferWindowPos function or by the most recent call to the DeferWindowPos function.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

The EndDeferWindowPos function sends the WM_WINDOWPOSCHANGING and WM_WINDOWPOSCHANGED messages to each window identified in the internal structure.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.126 EndDialog

The EndDialog function destroys a modal dialog box, causing the system to end any processing for the dialog box.

```
EndDialog: procedure
(
    hDlg            :dword;
    nResult         :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__EndDialog@8" );
```

Parameters

hDlg

[in] Handle to the dialog box to be destroyed.

nResult

[in] Specifies the value to be returned to the application from the function that created the dialog box.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

Dialog boxes created by the DialogBox, DialogBoxParam, DialogBoxIndirect, and DialogBoxIndirectParam functions must be destroyed using the EndDialog function. An application calls EndDialog from within the dia-

log box procedure; the function must not be used for any other purpose.

A dialog box procedure can call EndDialog at any time, even during the processing of the WM_INITDIALOG message. If your application calls the function while WM_INITDIALOG is being processed, the dialog box is destroyed before it is shown and before the input focus is set.

EndDialog does not destroy the dialog box immediately. Instead, it sets a flag and allows the dialog box procedure to return control to the system. The system checks the flag before attempting to retrieve the next message from the application queue. If the flag is set, the system ends the message loop, destroys the dialog box, and uses the value in nResult as the return value from the function that created the dialog box.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.127 EndMenu

The EndMenu function ends the calling thread's active menu.

```
EndMenu: procedure;
    @stdcall;
    @returns( "eax" );
    @external( "__imp__EndMenu@0" );
```

Parameters

This function has no parameters.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

If a platform does not support EndMenu, send the owner of the active menu a WM_CANCELMODE message.

Requirements

Windows NT/2000: Requires Windows 2000 or later.

Windows 95/98: Requires Windows 98 or later.

3.128 EndPaint

The EndPaint function marks the end of painting in the specified window. This function is required for each call to the BeginPaint function, but only after painting is complete.

```
EndPaint: procedure
(
        hWnd        :dword;
    var lpPaint     :PAINTSTRUCT
);
    @stdcall;
```

```
    @returns( "eax" );
    @external( "__imp__EndPaint@8" );
```

Parameters

hWnd

[in] Handle to the window that has been repainted.

lpPaint

[in] Pointer to a PAINTSTRUCT structure that contains the painting information retrieved by BeginPaint.

Return Values

The return value is always nonzero.

Remarks

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

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.129 EnumChildWindows

The EnumChildWindows function enumerates the child windows that belong to the specified parent window by passing the handle to each child window, in turn, to an application-defined callback function. EnumChildWindows continues until the last child window is enumerated or the callback function returns FALSE.

```
EnumChildWindows: procedure
(
    hWndParent      :dword;
    lpEnumFunc      :WNDENUMPROC;
    _lParam         :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__EnumChildWindows@12" );
```

Parameters

hWndParent

[in] Handle to the parent window whose child windows are to be enumerated. If this parameter is NULL, this function is equivalent to EnumWindows.

Windows 95/98: hWndParent cannot be NULL.

lpEnumFunc

[in] Pointer to an application-defined callback function. For more information, see EnumChildProc.

lParam

[in] Specifies an application-defined value to be passed to the callback function.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

If a child window has created child windows of its own, EnumChildWindows enumerates those windows as well.

A child window that is moved or repositioned in the Z order during the enumeration process will be properly enumerated. The function does not enumerate a child window that is destroyed before being enumerated or that is created during the enumeration process.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.130 EnumClipboardFormats

The EnumClipboardFormats function enumerates the data formats currently available on the clipboard.

Clipboard data formats are stored in an ordered list. To perform an enumeration of clipboard data formats, you make a series of calls to the EnumClipboardFormats function. For each call, the format parameter specifies an available clipboard format, and the function returns the next available clipboard format.

```
EnumClipboardFormats: procedure
(
    format          :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__EnumClipboardFormats@4" );
```

Parameters

format

[in] Specifies a clipboard format that is known to be available.

To start an enumeration of clipboard formats, set format to zero. When format is zero, the function retrieves the first available clipboard format. For subsequent calls during an enumeration, set format to the result of the previous EnumClipboardFormat call.

Return Values

If the function succeeds, the return value is the clipboard format that follows the specified format. In other words, the next available clipboard format.

If the function fails, the return value is zero. To get extended error information, call GetLastError. If the clipboard is not open, the function fails.

If there are no more clipboard formats to enumerate, the return value is zero. In this case, the GetLastError function returns the value NO_ERROR. This lets you distinguish between function failure and the end of enumeration.

Remarks

You must open the clipboard before enumerating its formats. Use the OpenClipboard function to open the clipboard. The EnumClipboardFormats function fails if the clipboard is not open.

The EnumClipboardFormats function enumerates formats in the order that they were placed on the clipboard. If you are copying information to the clipboard, add clipboard objects in order from the most descriptive clipboard

format to the least descriptive clipboard format. If you are pasting information from the clipboard, retrieve the first clipboard format that you can handle. That will be the most descriptive clipboard format that you can handle.

The system provides automatic type conversions for certain clipboard formats. In the case of such a format, this function enumerates the specified format, then enumerates the formats to which it can be converted. For more information, see Standard Clipboard Formats and Synthesized Clipboard Formats.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.131 EnumDesktopWindows

The EnumDesktopWindows function enumerates all top-level windows on a desktop by passing a handle to each window, in turn, to an application-defined callback function.

```
EnumDesktopWindows: procedure
(
    hDesktop        :dword;
    lpfn            :WNDENUMPROC;
    _lParam         :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__EnumDesktopWindows@12" );
```

Parameters

hDesktop

[in] Handle to the desktop whose top-level windows are to be enumerated. The CreateDesktop, OpenDesktop, and GetThreadDesktop functions return a desktop handle. If this parameter is NULL, the current desktop is used.

lpfn

[in] Pointer to an application-defined EnumWindowsProc callback function.

lParam

[in] Specifies an application-defined value to be passed to the callback function.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Windows 2000: If there are no windows on the desktop, GetLastError returns ERROR_INVALID_HANDLE.

Remarks

The EnumDesktopWindows function repeatedly invokes the lpfn callback function until the last top-level window is enumerated or the callback function returns FALSE.

Requirements

Windows NT/2000: Requires Windows NT 3.51 or later.

Windows 95/98: Unsupported.

3.132 EnumDesktops

The EnumDesktops function enumerates all desktops in the window station assigned to the calling process. The function does so by passing the name of each desktop, in turn, to an application-defined callback function.

```
EnumDesktops: procedure
(
    hwinsta         :dword;
    lpEnumFunc      :DESKTOPENUMPROC;
    _lParam         :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__EnumDesktopsA@12" );
```

Parameters

hwinsta

[in] Handle to the window station whose desktops are to be enumerated. The CreateWindowStation, GetProcessWindowStation, and OpenWindowStation functions return a window station handle.

lpEnumFunc

[in] Pointer to an application-defined EnumDesktopProc callback function.

lParam

[in] Specifies an application-defined value to be passed to the callback function.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

The calling process must have WINSTA_ENUMDESKTOPS access to the window station. The EnumDesktops function enumerates only those desktops for which the calling process has DESKTOP_ENUMERATE access.

The EnumDesktops function repeatedly invokes the lpEnumFunc callback function until the last desktop is enumerated or the callback function returns FALSE.

Requirements

Windows NT/2000: Requires Windows NT 3.51 or later.

Windows 95/98: Unsupported.

3.133 EnumDisplayMonitors

The EnumDisplayMonitors function enumerates display monitors (including invisible pseudo-monitors associated with the mirroring drivers) that intersect a region formed by the intersection of a specified clipping rectangle and the visible region of a device context. EnumDisplayMonitors calls an application-defined MonitorEnumProc callback function once for each monitor that is enumerated. Note that GetSystemMetrics(SM_CMONITORS) counts only the display monitors.

```
EnumDisplayMonitors: procedure
(
        hdc         :dword;
    var lprcClip    :CRECT;
        lpfnEnum    :MONITORENUMPROC;
        dwData      :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__EnumDisplayMonitors@16" );
```

Parameters

hdc

[in] Handle to a display device context that defines the visible region of interest.

If this parameter is NULL, the hdcMonitor parameter passed to the callback function will be NULL, and the visible region of interest is the virtual screen that encompasses all the displays on the desktop.

lprcClip

[in] Pointer to a RECT structure that specifies a clipping rectangle. The region of interest is the intersection of the clipping rectangle with the visible region specified by hdc.

If hdc is non-NULL, the coordinates of the clipping rectangle are relative to the origin of the hdc. If hdc is NULL, the coordinates are virtual-screen coordinates.

This parameter can be NULL if you don't want to clip the region specified by hdc.

lpfnEnum

[in] Pointer to a MonitorEnumProc application-defined callback function.

dwData

[in] Application-defined data that EnumDisplayMonitors passes directly to the MonitorEnumProc function.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Windows NT/2000: To get extended error information, call GetLastError.

Remarks

There are two reasons to call the EnumDisplayMonitors function:

- You want to draw optimally into a device context that spans several display monitors, and the monitors have different color formats.
- You want to obtain a handle and position rectangle for one or more display monitors.

To determine whether all the display monitors in a system share the same color format, call GetSystemMetrics(SM_SAMEDISPLAYFORMAT).

You do not need to use the EnumDisplayMonitors function when a window spans display monitors that have different color formats. You can continue to paint under the assumption that the entire screen has the color properties of the primary monitor. Your windows will look fine. EnumDisplayMonitors just lets you make them look better.

Setting the hdc parameter to NULL lets you use the EnumDisplayMonitors function to obtain a handle and position rectangle for one or more display monitors. The following table shows how the four combinations of NULL and non-NULL hdc and lprcClip values affect the behavior of the EnumDisplayMonitors function.

hdc	lprcRect	behavior
NULL	NULL	Enumerates all display monitors. The callback function receives a NULL HDC.
NULL	non-NULL	Enumerates all display monitors that intersect the clipping rectangle. Use virtual screen coordinates for the clipping rectangle. The callback function receives a NULL HDC.
non-NULL	NULL	Enumerates all display monitors that intersect the visible region of the device context. The callback function receives a handle to a DC for the specific display monitor.
non-NULL	non-NULL	Enumerates all display monitors that intersect the visible region of the device context and the clipping rectangle. Use device context coordinates for the clipping rectangle. The callback function receives a handle to a DC for the specific display monitor.

Examples

To paint in response to a WM_PAINT message, using the capabilities of each monitor, you can use code like this in a window procedure:

```
case WM_PAINT:
  hdc = BeginPaint(hwnd, &ps);
  EnumDisplayMonitors(hdc, NULL, MyPaintEnumProc, 0);
  EndPaint(hwnd, &ps);
```

To paint the top half of a window using the capabilities of each monitor, you can use code like this:

```
GetClientRect(hwnd, &rc);
rc.bottom = (rc.bottom - rc.top) / 2;
hdc = GetDC(hwnd);
EnumDisplayMonitors(hdc, &rc, MyPaintEnumProc, 0);
ReleaseDC(hwnd, hdc);
```

To paint the entire virtual screen optimally for each display monitor, you can use code like this:

```
hdc = GetDC(NULL);
EnumDisplayMonitors(hdc, NULL, MyPaintScreenEnumProc, 0);
ReleaseDC(NULL, hdc);
```

To get information about all of the display monitors, use code like this:

```
EnumDisplayMonitors(NULL, NULL, MyInfoEnumProc, 0);
```

Requirements

Windows NT/2000: Requires Windows 2000 or later.

Windows 95/98: Requires Windows 98 or later.

3.134 EnumDisplaySettings

The EnumDisplaySettings function retrieves information about one of the graphics modes for a display device. To retrieve information for all the graphics modes of a display device, make a series of calls to this function.

```
EnumDisplaySettings: procedure
(
        lpszDeviceName  :string;
        uModeNum        :dword;
    var lpDevMode       :DEVMODE
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__EnumDisplaySettingsA@12" );
```

Parameters

lpszDeviceName

[in] Pointer to a null-terminated string that specifies the display device about whose graphics mode the function will obtain information.

This parameter is either NULL or a DISPLAY_DEVICE.DeviceName returned from EnumDisplayDevices. A NULL value specifies the current display device on the computer on which the calling thread is running.

Windows 95: lpszDeviceName must be NULL.

iModeNum

[in] Specifies the type of information to retrieve. This value can be a graphics mode index or one of the following values.

Value	Meaning
ENUM_CURRENT_SETTINGS	Retrieve the current settings for the display device.
ENUM_REGISTRY_SETTINGS	Retrieve the settings for the display device that are currently stored in the registry.

Graphics mode indexes start at zero. To obtain information for all of a display device's graphics modes, make a series of calls to EnumDisplaySettings, as follows: Set iModeNum to zero for the first call, and increment iModeNum by one for each subsequent call. Continue calling the function until the return value is zero.

When you call EnumDisplaySettings with iModeNum set to zero, the operating system initializes and caches information about the display device. When you call EnumDisplaySettings with iModeNum set to a non-zero value, the function returns the information that was cached the last time the function was called with iModeNum set to zero.

lpDevMode

[out] Pointer to a DEVMODE structure into which the function stores information about the specified graphics mode. Before calling EnumDisplaySettings, set the dmSize member to sizeof(DEVMODE), and set the dmDriverExtra member to indicate the size, in bytes, of the additional space available to receive private driver data.

The EnumDisplaySettings function sets values for the following five DEVMODE members:

- dmBitsPerPel
- dmPelsWidth
- dmPelsHeight
- dmDisplayFlags
- dmDisplayFrequency

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Windows NT/2000: To get extended error information, call GetLastError.

Remarks

The function fails if iModeNum is greater than the index of the display device's last graphics mode. As noted in the description of the iModeNum parameter, you can use this behavior to enumerate all of a display device's graphics modes.

Requirements

Windows NT/2000: Requires Windows NT 3.51 or later.

Windows 95/98: Requires Windows 95 or later.

3.135 EnumDisplaySettingsEx

The EnumDisplaySettingsEx function retrieves information about one of the graphics modes for a display device. To retrieves information for all the graphics modes for a display device, make a series of calls to this function.

This function differs from EnumDisplaySettings in that there is a dwFlags parameter.

```
EnumDisplaySettingsEx: procedure
(
        lpszDeviceName  :string;
        uModeNum        :dword;
    var lpDevMode       :DEVMODE;
        dwFlags         :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__EnumDisplaySettingsExA@16" );
```

Parameters

lpszDeviceName

[in] Pointer to a null-terminated string that specifies the display device about which graphics mode the function will obtain information.

This parameter is either NULL or aDISPLAY_DEVICE.DeviceName returned from EnumDisplayDevices. A NULL value specifies the current display device on the computer that the calling thread is running on.

iModeNum

[in] Indicates the type of information to retrieve. This value can be a graphics mode index or one of the following values.

Value	Meaning
ENUM_CURRENT_SETTINGS	Retrieve the current settings for the display device.
ENUM_REGISTRY_SETTINGS	Retrieve the settings for the display device that are currently stored in the registry.

Graphics mode indexes start at zero. To obtain information for all of a display device's graphics modes, make a series of calls to EnumDisplaySettingsEx, as follows: Set iModeNum to zero for the first call, and increment

iModeNum by one for each subsequent call. Continue calling the function until the return value is zero. When you call EnumDisplaySettingsEx with iModeNum set to zero, the operating system initializes and caches information about the display device. When you call EnumDisplaySettingsEx with iModeNum set to a nonzero value, the function returns the information that was cached the last time the function was called with iModeNum set to zero.

lpDevMode

[out] Pointer to a DEVMODE structure into which the function stores information about the specified graphics mode. Before calling EnumDisplaySettingsEx, set the dmSize member to sizeof(DEVMODE), and set the dmDriverExtra member to indicate the size, in bytes, of the additional space available to receive private driver data.

The EnumDisplaySettingsEx function sets values for the following five DEVMODE members:

- dmBitsPerPel
- dmPelsWidth
- dmPelsHeight
- dmDisplayFlags
- dmDisplayFrequency

dwFlags

[in] This parameter can be the following value.

Value	Meaning
EDS_RAWMODE	If set, the function will return all graphics modes reported by the adapter driver, regardless of monitor capabilities. Otherwise, it will only return modes that are compatible with current monitors.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

To get extended error information, call GetLastError.

Remarks

The function fails if iModeNum is greater than the index of the display device's last graphics mode. As noted in the description of the iModeNum parameter, you can use this behavior to enumerate all of a display device's graphics modes.

Requirements

Windows NT/2000: Requires Windows 2000 or later.

Windows 95/98: Requires Windows 98 or later.

3.136 EnumProps

The EnumProps function enumerates all entries in the property list of a window by passing them, one by one, to the specified callback function. EnumProps continues until the last entry is enumerated or the callback function returns FALSE.

To pass application-defined data to the callback function, use the EnumPropsEx function.

```
EnumProps: procedure
(
    hWnd            :dword;
    lpEnumFunc      :PROPENUMPROC
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__EnumPropsA@8" );
```

Parameters

hWnd

[in] Handle to the window whose property list is to be enumerated.

lpEnumFunc

[in] Pointer to the callback function. For more information about the callback function, see the PropEnumProc function.

Return Values

The return value specifies the last value returned by the callback function. It is -1 if the function did not find a property for enumeration.

Remarks

An application can remove only those properties it has added. It must not remove properties added by other applications or by the system itself.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.137 EnumPropsEx

The EnumPropsEx function enumerates all entries in the property list of a window by passing them, one by one, to the specified callback function. EnumPropsEx continues until the last entry is enumerated or the callback function returns FALSE.

```
EnumPropsEx: procedure
(
    hWnd            :dword;
    lpEnumFunc      :PROPENUMPROC;
    _lParam         :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__EnumPropsExA@12" );
```

Parameters

hWnd

[in] Handle to the window whose property list is to be enumerated.

lpEnumFunc

[in] Pointer to the callback function. For more information about the callback function, see the PropEnumProcEx function.

lParam

[in] Contains application-defined data to be passed to the callback function.

Return Values

The return value specifies the last value returned by the callback function. It is -1 if the function did not find a property for enumeration.

Remarks

An application can remove only those properties it has added. It must not remove properties added by other applications or by the system itself.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.138 EnumThreadWindows

The EnumThreadWindows function enumerates all nonchild windows associated with a thread by passing the handle to each window, in turn, to an application-defined callback function. EnumThreadWindows continues until the last window is enumerated or the callback function returns FALSE. To enumerate child windows of a particular window, use the EnumChildWindows function.

```
EnumThreadWindows: procedure
(
    dwThreadId      :dword;
    lpfn            :WNDENUMPROC;
    _lParam         :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__EnumThreadWindows@12" );
```

Parameters

dwThreadId

[in] Identifies the thread whose windows are to be enumerated.

lpfn

[in] Pointer to an application-defined callback function. For more information, see EnumThreadWndProc.

lParam

[in] Specifies an application-defined value to be passed to the callback function.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.139 EnumWindowStations

The EnumWindowStations function enumerates all window stations in the system by passing the name of each window station, in turn, to an application-defined callback function.

```
EnumWindowStations: procedure
(
    lpEnumFunc      :WINSTAENUMPROC;
    _lParam         :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__EnumWindowStationsA@8" );
```

Parameters

lpEnumFunc

[in] Pointer to an application-defined EnumWindowStationProc callback function.

lParam

[in] Specifies an application-defined value to be passed to the callback function.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

The EnumWindowStations function enumerates only those window stations for which the calling process has WINSTA_ENUMERATE access.

EnumWindowStations repeatedly invokes the lpEnumFunc callback function until the last window station is enumerated or the callback function returns FALSE.

Requirements

Windows NT/2000: Requires Windows NT 3.51 or later.

Windows 95/98: Unsupported.

3.140 EnumWindows

The EnumWindows function enumerates all top-level windows on the screen by passing the handle to each window, in turn, to an application-defined callback function. EnumWindows continues until the last top-level window is enumerated or the callback function returns FALSE.

```
EnumWindows: procedure
(
    lpEnumFunc      :WINENUMPROC;
    _lParam         :dword
);
    @stdcall;
```

```
    @returns( "eax" );
    @external( "__imp__EnumWindows@8" );
```

Parameters

lpEnumFunc

[in] Pointer to an application-defined callback function. For more information, see EnumWindowsProc.

lParam

[in] Specifies an application-defined value to be passed to the callback function.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

If EnumWindowsProc returns zero, the return value is also zero.

Remarks

The EnumWindows function does not enumerate child windows, with the exception of a few top-level windows owned by the system that have the WS_CHILD style.

This function is more reliable than calling the GetWindow function in a loop. An application that calls GetWindow to perform this task risks being caught in an infinite loop or referencing a handle to a window that has been destroyed.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.141 EqualRect

The EqualRect function determines whether the two specified rectangles are equal by comparing the coordinates of their upper-left and lower-right corners.

```
EqualRect: procedure
(
    var lprc1       :RECT;
    var lprc2       :RECT
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__EqualRect@8" );
```

Parameters

lprc1

[in] Pointer to a RECT structure that contains the logical coordinates of the first rectangle.

lprc2

[in] Pointer to a RECT structure that contains the logical coordinates of the second rectangle.

Return Values

If the two rectangles are identical, the return value is nonzero.

If the two rectangles are not identical, the return value is zero.

Windows NT/ 2000: To get extended error information, call GetLastError.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.142 ExcludeUpdateRgn

The ExcludeUpdateRgn function prevents drawing within invalid areas of a window by excluding an updated region in the window from a clipping region.

```
ExcludeUpdateRgn: procedure
(
    hdc         :dword;
    hWnd        :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__ExcludeUpdateRgn@8" );
```

Parameters

hDC

[in] Handle to the device context associated with the clipping region.

hWnd

[in] Handle to the window to update.

Return Values

The return value specifies the complexity of the excluded region; it can be any one of the following values.

Value	Meaning
COMPLEXREGION	Region consists of more than one rectangle.
ERROR	An error occurred.
NULLREGION	Region is empty.
SIMPLEREGION	Region is a single rectangle.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.143 ExitWindowsEx

The ExitWindowsEx function either logs off the current user, shuts down the system, or shuts down and restarts

the system. It sends the WM_QUERYENDSESSION message to all applications to determine if they can be terminated.

```
ExitWindowsEx: procedure
(
    uFlags      :dword;
    dwReserved  :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__ExitWindowsEx@8" );
```

Parameters

uFlags

[in] Specifies the type of shutdown. This parameter must include one of the following values.

Value	Meaning
EWX_LOGOFF	Shuts down all processes running in the security context of the process that called the function. Then it logs the user off.
EWX_POWEROFF	Shuts down the system and turns off the power. The system must support the power-off feature. Windows NT/2000: The calling process must have the SE_SHUTDOWN_NAME privilege. For more information, see the following Remarks section.
EWX_REBOOT	Shuts down the system and then restarts the system. Windows NT/2000: The calling process must have the SE_SHUTDOWN_NAME privilege. For more information, see the following Remarks section.
EWX_SHUTDOWN	Shuts down the system to a point at which it is safe to turn off the power. All file buffers have been flushed to disk, and all running processes have stopped. Windows NT/2000: The calling process must have the SE_SHUTDOWN_NAME privilege. For more information, see the following Remarks section.

This parameter can optionally include the following values.

Value	Meaning
EWX_FORCE	Forces processes to terminate. When this flag is set, the system does not send the WM_QUERYENDSESSION and WM_ENDSESSION messages. This can cause the applications to lose data. Therefore, you should only use this flag in an emergency.
EWX_FORCEIFHUNG	Windows 2000: Forces processes to terminate if they do not respond to the WM_QUERYENDSESSION or WM_ENDSESSION message. This flag is ignored if EWX_FORCE is used.

dwReserved

[in] Reserved; this parameter is ignored.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

The ExitWindowsEx function returns as soon as it has initiated the shutdown. The shutdown or logoff then proceeds asynchronously.

During a shutdown or log-off operation, applications that are shut down are allowed a specific amount of time to respond to the shutdown request. If the time expires, the system displays a dialog box that allows the user to forcibly shut down the application, to retry the shutdown, or to cancel the shutdown request. If the EWX_FORCE value is specified, the system always forces applications to close and does not display the dialog box.

Windows 2000: If the EWX_FORCEIFHUNG value is specified, the system forces hung applications to close and does not display the dialog box.

Windows 95/98: Because of the design of the shell, calling ExitWindowsEx with EWX_FORCE fails to completely log off the user (the system terminates the applications and displays the Enter Windows Password dialog box, however, the user's desktop remains.) To log off the user forcibly, terminate the Explorer process before calling ExitWindowsEx with EWX_LOGOFF and EWX_FORCE.

Console processes receive a separate notification message, CTRL_SHUTDOWN_EVENT or CTRL_LOGOFF_EVENT, as the situation warrants. A console process routes these messages to its HandlerRoutine function. ExitWindowsEx sends these notification messages asynchronously; thus, an application cannot assume that the console notification messages have been handled when a call to ExitWindowsEx returns.

Windows NT/2000: To shut down or restart the system, the calling process must use the AdjustTokenPrivileges function to enable the SE_SHUTDOWN_NAME privilege. For more information about security privileges, see Privileges.

Windows 95/98: ExitWindowEx does not work from a console application, as it does on Windows NT/Windows 2000.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.144 FillRect

The FillRect function fills a rectangle by using the specified brush. This function includes the left and top borders, but excludes the right and bottom borders of the rectangle.

```
FillRect: procedure
(
        hDC         :dword;
    var lprc        :RECT;
        hbr         :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__FillRect@12" );
```

Parameters

hDC

[in] Handle to the device context.

lprc

[in] Pointer to a RECT structure that contains the logical coordinates of the rectangle to be filled.

hbr

[in] Handle to the brush used to fill the rectangle.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Windows NT/ 2000: To get extended error information, call GetLastError.

Remarks

The brush identified by the hbr parameter may be either a handle to a logical brush or a color value. If specifying a handle to a logical brush, call one of the following functions to obtain the handle: CreateHatchBrush, CreatePatternBrush, or CreateSolidBrush. Additionally, you may retrieve a handle to one of the stock brushes by using the GetStockObject function. If specifying a color value for the hbr parameter, it must be one of the standard system colors (the value 1 must be added to the chosen color). For example:

FillRect(hdc, &rect, (HBRUSH) (COLOR_WINDOW+1));

For a list of all the standard system colors, see GetSysColor.

When filling the specified rectangle, FillRect does not include the rectangle's right and bottom sides. GDI fills a rectangle up to, but not including, the right column and bottom row, regardless of the current mapping mode.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.145 FindWindow

The FindWindow function retrieves a handle to the top-level window whose class name and window name match the specified strings. This function does not search child windows. This function does not perform a case-sensitive search.

To search child windows, beginning with a specified child window, use the FindWindowEx function.

```
FindWindow: procedure
(
    lpClassName     :string;
    lpWindowName    :string
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__FindWindowA@8" );
```

Parameters

lpClassName

[in] Pointer to a null-terminated string that specifies the class name or a class atom created by a previous call to the RegisterClass or RegisterClassEx function. The atom must be in the low-order word of lpClassName; the high-order word must be zero.

If lpClassName is a string, it specifies the window class name. The class name can be any name registered

with RegisterClass or RegisterClassEx, or any of the predefined control-class names.

lpWindowName

[in] Pointer to a null-terminated string that specifies the window name (the window's title). If this parameter is NULL, all window names match.

Return Values

If the function succeeds, the return value is a handle to the window that has the specified class name and window name.

If the function fails, the return value is NULL. To get extended error information, call GetLastError.

Remarks

If the lpWindowName parameter is not NULL, FindWindow calls the GetWindowText function to retrieve the window name for comparison. For a description of a potential problem that can arise, see the Remarks for GetWindowText.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.146 FindWindowEx

The FindWindowEx function retrieves a handle to a window whose class name and window name match the specified strings. The function searches child windows, beginning with the one following the specified child window. This function does not perform a case-sensitive search.

```
FindWindowEx: procedure
(
    hwndParent      :dword;
    hwndChildAfter  :dword;
    lpszClass       :string;
    lpszWindow      :string
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__FindWindowExA@16" );
```

Parameters

hwndParent

[in] Handle to the parent window whose child windows are to be searched.

If hwndParent is NULL, the function uses the desktop window as the parent window. The function searches among windows that are child windows of the desktop.

Windows 2000: If hwndParent is HWND_MESSAGE, the function searches all message-only windows.

hwndChildAfter

[in] Handle to a child window. The search begins with the next child window in the Z order. The child window must be a direct child window of hwndParent, not just a descendant window.

If hwndChildAfter is NULL, the search begins with the first child window of hwndParent.

Note that if both hwndParent and hwndChildAfter are NULL, the function searches all top-level and mes-

sage-only windows.

lpszClass

[in] Pointer to a null-terminated string that specifies the class name or a class atom created by a previous call to the RegisterClass or RegisterClassEx function. The atom must be placed in the low-order word of lpszClass; the high-order word must be zero.

If lpszClass is a string, it specifies the window class name. The class name can be any name registered with RegisterClass or RegisterClassEx, or any of the predefined control-class names.

lpszWindow

[in] Pointer to a null-terminated string that specifies the window name (the window's title). If this parameter is NULL, all window names match.

Return Values

If the function succeeds, the return value is a handle to the window that has the specified class and window names.

If the function fails, the return value is NULL. To get extended error information, call GetLastError.

Remarks

If the lpWindowName parameter is not NULL, FindWindowEx calls the GetWindowText function to retrieve the window name for comparison. For a description of a potential problem that can arise, see the Remarks for GetWindowText.

Requirements

Windows NT/2000: Requires Windows NT 4.0 or later.

Windows 95/98: Requires Windows 95 or later.

3.147 FlashWindow

The FlashWindow function flashes the specified window one time. It does not change the active state of the window.

To flash the window a specified number of times, use the FlashWindowEx function.

```
FlashWindow: procedure
(
    hWnd            :dword;
    bInvert         :boolean;
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__FlashWindow@8" );
```

Parameters

hWnd

[in] Handle to the window to be flashed. The window can be either open or minimized.

bInvert

[in] Specifies whether the window is to be flashed or returned to its original state. The window is flashed from one state to the other if this parameter is TRUE. If it is FALSE, the window is returned to its original state (either active or inactive). When an application is minimized, if this parameter is TRUE, the taskbar window

button flashes active/inactive. If it is FALSE, the taskbar window button flashes inactive, meaning that it does not change colors. It flashes, as if it were being redraw, but it does not provide the visual invert clue to the user.

Return Values

The return value specifies the window's state before the call to the FlashWindow function. If the window caption was drawn as active before the call, the return value is nonzero. Otherwise, the return value is zero.

Remarks

Flashing a window means changing the appearance of its caption bar as if the window were changing from inactive to active status, or vice versa. (An inactive caption bar changes to an active caption bar; an active caption bar changes to an inactive caption bar.)

Typically, a window is flashed to inform the user that the window requires attention but that it does not currently have the keyboard focus.

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

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.148 FrameRect

The FrameRect function draws a border around the specified rectangle by using the specified brush. The width and height of the border are always one logical unit.

```
FrameRect: procedure
(
        hDC         :dword;
    var lprc        :RECT;
        hbr         :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__FrameRect@12" );
```

Parameters

hDC

[in] Handle to the device context in which the border is drawn.

lprc

[in] Pointer to a RECT structure that contains the logical coordinates of the upper-left and lower-right corners of the rectangle.

hbr

[in] Handle to the brush used to draw the border.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Windows NT/ 2000: To get extended error information, call GetLastError.

Remarks

The brush identified by the hbr parameter must have been created by using the CreateHatchBrush, CreatePattern-Brush, or CreateSolidBrush function, or retrieved by using the GetStockObject function.

If the bottom member of the RECT structure is less than or equal to the top member, or if the right member is less than or equal to the left member, the function does not draw the rectangle.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.149 FreeDDElParam

The FreeDDElParam function frees the memory specified by the lParam parameter of a posted DDE message. An application receiving a posted DDE message should call this function after it has used the UnpackDDElParam function to unpack the lParam value.

```
FreeDDElParam: procedure
(
    _msg            :dword;
    _lParam         :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__FreeDDElParam@8" );
```

Parameters

msg

[in] Specifies the posted DDE message.

lParam

[in] Specifies the lParam parameter of the posted DDE message.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Remarks

An application should call this function only for posted DDE messages.

This function frees the memory specified by the lParam parameter. It does not free the contents of lParam.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.150 GetActiveWindow

The GetActiveWindow function retrieves the window handle to the active window attached to the calling thread's message queue.

```
GetActiveWindow: procedure;
    @stdcall;
    @returns( "eax" );
    @external( "__imp__GetActiveWindow@0" );
```

Parameters

This function has no parameters.

Return Values

The return value is the handle to the active window attached to the calling thread's message queue. Otherwise, the return value is NULL.

Remarks

To get the handle to the foreground window, you can use GetForegroundWindow.

Windows 98 and Windows NT 4.0 SP3 and later: To get the window handle to the active window in the message queue for another thread, use GetGUIThreadInfo.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.151 GetAltTabInfoW

The GetAltTabInfoW function retrieves status information for the specified window if it is the application-switching (ALT+TAB) window.

```
GetAltTabInfoW: procedure
(
        hwnd        :dword;
        iItem       :dword;
    var pati        :dword;
    var pszItemText :var;
        cchItemText :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__GetAltTabInfo@20" );
```

Parameters

hwnd

[in] Handle to the window for which status information will be retrieved. This window must be the application-switching window.

iItem

[in] Specifies the index of the icon in the application-switching window. If the pszItemText parameter is not

NULL, the name of the item is copied to the pszItemText string. If this parameter is –1, the name of the item is not copied.

pati

[out] Pointer to an ALTTABINFO structure to receive the status information.

pszItemText

[out] Pointer to a string that receives the name of the item. If this parameter is NULL, the name of the item is not copied.

cchItemText

[in] Specifies the size, in TCHARs, of the pszItemText buffer.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

The application-switching window enables you to switch to the most recently used application window. To display the application-switching window, press ALT+TAB. To select an application from the list, continue to hold ALT down and press TAB to move through the list. Add SHIFT to reverse direction through the list.

Requirements

Windows NT/2000: Requires Windows NT 4.0 SP6 or later.

Windows 95/98: Requires Windows 98 or later.

3.152 GetAltTabInfo

The GetAltTabInfo function retrieves status information for the specified window if it is the application-switching (ALT+TAB) window.

```
GetAltTabInfo: procedure
(
        hwnd        :dword;
        iItem       :dword;
    var pati        :ALTTABLINFO;
    var pszItemText :var;
        cchItemText :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__GetAltTabInfoA@20" );
```

Parameters

hwnd

[in] Handle to the window for which status information will be retrieved. This window must be the application-switching window.

iItem

[in] Specifies the index of the icon in the application-switching window. If the pszItemText parameter is not NULL, the name of the item is copied to the pszItemText string. If this parameter is –1, the name of the item is not copied.

pati

[out] Pointer to an ALTTABINFO structure to receive the status information.

pszItemText

[out] Pointer to a string that receives the name of the item. If this parameter is NULL, the name of the item is not copied.

cchItemText

[in] Specifies the size, in TCHARs, of the pszItemText buffer.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

The application-switching window enables you to switch to the most recently used application window. To display the application-switching window, press ALT+TAB. To select an application from the list, continue to hold ALT down and press TAB to move through the list. Add SHIFT to reverse direction through the list.

Requirements

Windows NT/2000: Requires Windows NT 4.0 SP6 or later.

Windows 95/98: Requires Windows 98 or later.

3.153 GetAncestor

The GetAncestor function retrieves the handle to the ancestor of the specified window.

```
GetAncestor: procedure
(
    hwnd            :dword;
    gaFlags         :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__GetAncestor@8" );
```

Parameters

hwnd

[in] Handle to the window whose ancestor is to be retrieved. If this parameter is the desktop window, the function returns NULL.

gaFlags

[in] Specifies the ancestor to be retrieved. This parameter can be one of the following values.

Value	Meaning
GA_PARENT	Retrieves the parent window. This does not include the owner, as it does with the GetParent function.
GA_ROOT	Retrieves the root window by walking the chain of parent windows.

GA_ROOTOWNER	Retrieves the owned root window by walking the chain of parent and owner windows returned by GetParent.

Return Values

The return value is the handle to the ancestor window.

Requirements

Windows NT/2000: Requires Windows NT 4.0 SP4 or later.

Windows 95/98: Requires Windows 98 or later.

3.154 GetAsyncKeyState

The GetAsyncKeyState function determines whether a key is up or down at the time the function is called, and whether the key was pressed after a previous call to GetAsyncKeyState.

```
GetAsyncKeyState: procedure
(
    vKey            :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__GetAsyncKeyState@4" );
```

Parameters

vKey

[in] Specifies one of 256 possible virtual-key codes. For more information, see Virtual-Key Codes.

Windows NT/2000: You can use left- and right-distinguishing constants to specify certain keys. See the Remarks section for further information.

Return Values

If the function succeeds, the return value specifies whether the key was pressed since the last call to GetAsyncKeyState, and whether the key is currently up or down. If the most significant bit is set, the key is down, and if the least significant bit is set, the key was pressed after the previous call to GetAsyncKeyState. However, you should not rely on this last behavior; for more information, see the Remarks. The return value is zero if a window in another thread or process currently has the keyboard focus.

Windows 95: Windows 95 does not support the left- and right-distinguishing constants. If you call GetAsyncKeyState with these constants, the return value is zero.

Remarks

The GetAsyncKeyState function works with mouse buttons. However, it checks on the state of the physical mouse buttons, not on the logical mouse buttons that the physical buttons are mapped to. For example, the call GetAsyncKeyState(VK_LBUTTON) always returns the state of the left physical mouse button, regardless of whether it is mapped to the left or right logical mouse button. You can determine the system's current mapping of physical mouse buttons to logical mouse buttons by calling

GetSystemMetrics(SM_SWAPBUTTON)

which returns TRUE if the mouse buttons have been swapped.

Although the least significant bit of the return value indicates whether the key has been pressed since the last query, due to the pre-emptive multitasking nature of Win32, another application can call GetAsyncKeyState and receive the "recently pressed" bit instead of your application. The behavior of the least significant bit of the return

value is retained strictly for compatibility with 16-bit Windows applications (which are non-preemptive) and should not be relied upon.

You can use the virtual-key code constants VK_SHIFT, VK_CONTROL, and VK_MENU as values for the vKey parameter. This gives the state of the SHIFT, CTRL, or ALT keys without distinguishing between left and right.

Windows NT/2000: You can use the following virtual-key code constants as values for vKey to distinguish between the left and right instances of those keys.

Code	Meaning
VK_LSHIFT	VK_RSHIFT
VK_LCONTROL	VK_RCONTROL
VK_LMENU	VK_RMENU

These left- and right-distinguishing constants are only available when you call the GetKeyboardState, SetKeyboardState, GetAsyncKeyState, GetKeyState, and MapVirtualKey functions.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.155 GetCapture

The GetCapture function retrieves a handle to the window (if any) that has captured the mouse. Only one window at a time can capture the mouse; this window receives mouse input whether or not the cursor is within its borders.

```
GetCapture: procedure;
    @stdcall;
    @returns( "eax" );
    @external( "__imp__GetCapture@0" );
```

Parameters

This function has no parameters.

Return Values

The return value is a handle to the capture window associated with the current thread. If no window in the thread has captured the mouse, the return value is NULL.

Remarks

A NULL return value means the current thread has not captured the mouse. However, it is possible that another thread or process has captured the mouse.

Windows 98 and Windows NT 4.0 SP3 and later: To get a handle to the capture window on another thread, use the GetGUIThreadInfo function.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.156 GetCaretBlinkTime

The GetCaretBlinkTime function returns the time required to invert the caret's pixels. The user can set this value.

```
GetCaretBlinkTime: procedure;
    @stdcall;
    @returns( "eax" );
    @external( "__imp__GetCaretBlinkTime@0" );
```

Parameters

This function has no parameters.

Return Values

If the function succeeds, the return value is the blink time, in milliseconds.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.157 GetCaretPos

The GetCaretPos function copies the caret's position to the specified POINT structure.

```
GetCaretPos: procedure
(
    var lpPoint     :POINT
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__GetCaretPos@4" );
```

Parameters

lpPoint

[out] Pointer to the POINT structure that is to receive the client coordinates of the caret.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

The caret position is always given in the client coordinates of the window that contains the caret.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.158 GetClassInfo

The GetClassInfo function retrieves information about a window class.

Note The GetClassInfo function has been superseded by the GetClassInfoEx function. You can still use GetClassInfo, however, if you do not need information about the class small icon.

```
GetClassInfo: procedure
(
        hInstance   :dword;
        lpClassName :string;
    var lpWndClass  :WNDCLASS
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__GetClassInfoA@12" );
```

Parameters

hInstance

[in] Handle to the instance of the application that created the class. To retrieve information about classes defined by the system (such as buttons or list boxes), set this parameter to NULL.

lpClassName

[in] Pointer to a null-terminated string containing the class name. The name must be that of a preregistered class or a class registered by a previous call to the RegisterClass or RegisterClassEx function.

Alternatively, this parameter can be an atom. If so, it must be a class atom created by a previous call to RegisterClass or RegisterClassEx. The atom must be in the low-order word of lpClassName; the high-order word must be zero.

lpWndClass

[out] Pointer to a WNDCLASS structure that receives the information about the class.

Return Values

If the function finds a matching class and successfully copies the data, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.159 GetClassInfoEx

The GetClassInfoEx function retrieves information about a window class, including a handle to the small icon associated with the window class. The GetClassInfo function does not retrieve a handle to the small icon.

```
GetClassInfoEx: procedure
(
        hinst       :dword;
        lpszClass   :string;
    var lpwcx       :WNDCLASSEX
);
```

```
    @stdcall;
    @returns( "eax" );
    @external( "__imp__GetClassInfoExA@12" );
```

Parameters

hinst

[in] Handle to the instance of the application that created the class. To retrieve information about classes defined by the system (such as buttons or list boxes), set this parameter to NULL.

lpszClass

[in] Pointer to a null-terminated string containing the class name. The name must be that of a preregistered class or a class registered by a previous call to the RegisterClass or RegisterClassEx function.

Alternatively, this parameter can be a class atom created by a previous call to RegisterClass or RegisterClassEx. The atom must be in the low-order word of lpszClass; the high-order word must be zero.

lpwcx

[out] Pointer to a WNDCLASSEX structure that receives the information about the class.

Return Values

If the function finds a matching class and successfully copies the data, the return value is nonzero.

If the function does not find a matching class and successfully copy the data, the return value is zero. To get extended error information, call GetLastError.

Remarks

Class atoms are created using the RegisterClass or RegisterClassEx function, not the GlobalAddAtom function.

Requirements

Windows NT/2000: Requires Windows NT 3.5 or later.

Windows 95/98: Requires Windows 95 or later.

3.160 GetClassLong

The GetClassLong function retrieves the specified 32-bit (long) value from the WNDCLASSEX structure associated with the specified window.

If you are retrieving a pointer or a handle, this function has been superseded by the GetClassLongPtr function. (Pointers and handles are 32 bits on 32-bit Windows and 64 bits on 64-bit Windows.) To write code that is compatible with both 32-bit and 64-bit versions of Windows, use GetClassLongPtr.

```
GetClassLong: procedure
(
    hWnd            :dword;
    nIndex          :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__GetClassLongA@8" );
```

Parameters

hWnd

[in] Handle to the window and, indirectly, the class to which the window belongs.

nIndex

[in] Specifies the 32-bit value to retrieve. To retrieve a 32-bit value from the extra class memory, specify the positive, zero-based byte offset of the value to be retrieved. Valid values are in the range zero through the number of bytes of extra class memory, minus four; for example, if you specified 12 or more bytes of extra class memory, a value of 8 would be an index to the third 32-bit integer. To retrieve any other value from the WNDCLASSEX structure, specify one of the following values.

Value	Action
GCW_ATOM	Retrieves an ATOM value that uniquely identifies the window class. This is the same atom that the RegisterClassEx function returns.
GCL_CBCLSEXTRA	Retrieves the size, in bytes, of the extra memory associated with the class.
GCL_CBWNDEXTRA	Retrieves the size, in bytes, of the extra window memory associated with each window in the class. For information on how to access this memory, see GetWindowLong.
GCL_HBRBACKGROUND	Retrieves a handle to the background brush associated with the class.
GCL_HCURSOR	Retrieves a handle to the cursor associated with the class.
GCL_HICON	Retrieves a handle to the icon associated with the class.
GCL_HICONSM	Retrieves a handle to the small icon associated with the class.
GCL_HMODULE	Retrieves a handle to the module that registered the class.
GCL_MENUNAME	Retrieves the address of the menu name string. The string identifies the menu resource associated with the class.
GCL_STYLE	Retrieves the window-class style bits.
GCL_WNDPROC	Retrieves the address of the window procedure, or a handle representing the address of the window procedure. You must use the CallWindowProc function to call the window procedure.

Return Values

If the function succeeds, the return value is the requested 32-bit value.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

Reserve extra class memory by specifying a nonzero value in the cbClsExtra member of the WNDCLASSEX structure used with the RegisterClassEx function.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.161 GetClassName

The GetClassName function retrieves the name of the class to which the specified window belongs.

```
GetClassName: procedure
(
    hWnd            :dword;
    lpClassName     :string;
    nMaxCount       :dword
);
```

```
    @stdcall;
    @returns( "eax" );
    @external( "__imp__GetClassNameA@12" );
```

Parameters

hWnd

[in] Handle to the window and, indirectly, the class to which the window belongs.

lpClassName

[out] Pointer to the buffer that is to receive the class name string.

nMaxCount

[in] Specifies the length, in TCHARs, of the buffer pointed to by the lpClassName parameter. The class name string is truncated if it is longer than the buffer.

Return Values

If the function succeeds, the return value is the number of TCHARs copied to the specified buffer.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.162 GetClassWord

The GetClassWord function retrieves the 16-bit (word) value at the specified offset into the extra class memory for the window class to which the specified window belongs.

Note This function is provided only for compatibility with 16-bit versions of Windows. The only offsets supported on 32-bit Windows are GCW_ATOM and GCW_HICONSM. Win32-based applications should use the GetClassLong function.

```
GetClassWord: procedure
(
    hWnd            :dword;
    nIndex          :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__GetClassWord@8" );
```

Parameters

hWnd

[in] Handle to the window and, indirectly, the class to which the window belongs.

nIndex

[in] Specifies the zero-based byte offset of the value to be retrieved. Valid values are in the range zero through the number of bytes of class memory, minus two; for example, if you specified 10 or more bytes of extra class memory, a value of eight would be an index to the fifth 16-bit integer. There is an additional valid value as shown in the following table.

Value	Action
GCW_ATOM	Retrieves an ATOM value that uniquely identifies the window class. This is the same atom that the RegisterClass or RegisterClassEx function returns.
GCW_HICONSM	Retrieves a handle to the small icon associated with the window.

Return Values

If the function succeeds, the return value is the requested 16-bit value.

If the function fails, the return value is otherwise, it is zero. To get extended error information, call GetLastError.

Remarks

Reserve extra class memory by specifying a nonzero value in the cbClsExtra member of the WNDCLASS structure used with the RegisterClass function.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.163 GetClientRect

The GetClientRect function retrieves the coordinates of a window's client area. The client coordinates specify the upper-left and lower-right corners of the client area. Because client coordinates are relative to the upper-left corner of a window's client area, the coordinates of the upper-left corner are (0,0).

```
GetClientRect: procedure
(
        hWnd        :dword;
    var lpRect      :RECT
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__GetClientRect@8" );
```

Parameters

hWnd

[in] Handle to the window whose client coordinates are to be retrieved.

lpRect

[out] Pointer to a RECT structure that receives the client coordinates. The left and top members are zero. The right and bottom members contain the width and height of the window.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.164 GetClipCursor

The GetClipCursor function retrieves the screen coordinates of the rectangular area to which the cursor is confined.

```
GetClipCursor: procedure
(
    var lpRect      :RECT
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__GetClipCursor@4" );
```

Parameters

lpRect

[out] Pointer to a RECT structure that receives the screen coordinates of the confining rectangle. The structure receives the dimensions of the screen if the cursor is not confined to a rectangle.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

The cursor is a shared resource. If an application confines the cursor with the ClipCursor function, it must later release the cursor by using ClipCursor before relinquishing control to another application.

The calling process must have WINSTA_READATTRIBUTES access to the window station.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.165 GetClipboardData

The GetClipboardData function retrieves data from the clipboard in a specified format. The clipboard must have been opened previously.

```
GetClipboardData: procedure
(
    uFormat         :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__GetClipboardData@4" );
```

Parameters

uFormat

[in] Specifies a clipboard format. For a description of the standard clipboard formats, see Standard Clipboard

Formats.

Return Values

If the function succeeds, the return value is the handle to a clipboard object in the specified format.

If the function fails, the return value is NULL. To get extended error information, call GetLastError.

Remarks

An application can enumerate the available formats in advance by using the EnumClipboardFormats function.

The clipboard controls the handle that the GetClipboardData function returns, not the application. The application should copy the data immediately. The application must not free the handle nor leave it locked. The application must not use the handle after the EmptyClipboard or CloseClipboard function is called, or after the SetClipboardData function is called with the same clipboard format.

The system performs implicit data format conversions between certain clipboard formats when an application calls the GetClipboardData function. For example, if the CF_OEMTEXT format is on the clipboard, a window can retrieve data in the CF_TEXT format. The format on the clipboard is converted to the requested format on demand. For more information, see Synthesized Clipboard Formats.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.166 GetClipboardFormatName

The GetClipboardFormatName function retrieves from the clipboard the name of the specified registered format. The function copies the name to the specified buffer.

```
GetClipboardFormatName: procedure
(
    format          :dword;
    lpszFormatName  :string;
    cchMaxCount     :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__GetClipboardFormatNameA@12" );
```

Parameters

format

[in] Specifies the type of format to be retrieved. This parameter must not specify any of the predefined clipboard formats.

lpszFormatName

[out] Pointer to the buffer that is to receive the format name.

cchMaxCount

[in] Specifies the maximum length, in TCHARs, of the string to be copied to the buffer. If the name exceeds this limit, it is truncated. For ANSI versions of the function this is the number of bytes, while for wide-character (Unicode) versions this is the number of characters.

Return Values

If the function succeeds, the return value is the length, in TCHARs, of the string copied to the buffer.

If the function fails, the return value is zero, indicating that the requested format does not exist or is predefined. To get extended error information, call GetLastError.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.167 GetClipboardOwner

The GetClipboardOwner function retrieves the window handle of the current owner of the clipboard.

```
GetClipboardOwner: procedure;
    @stdcall;
    @returns( "eax" );
    @external( "__imp__GetClipboardOwner@0" );
```

Parameters

This function has no parameters.

Return Values

If the function succeeds, the return value is the handle to the window that owns the clipboard.

If the clipboard is not owned, the return value is NULL. To get extended error information, call GetLastError.

Remarks

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

In general, the clipboard owner is the window that last placed data in clipboard. The EmptyClipboard function assigns clipboard ownership.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.168 GetClipboardSequenceNumber

The GetClipboardSequenceNumber function retrieves the clipboard sequence number for the current window station.

```
GetClipboardSequenceNumber: procedure;
    @stdcall;
    @returns( "eax" );
    @external( "__imp__GetClipboardSequenceNumber@0" );
```

Parameters

This function has no parameters.

Return Values

The return value is the clipboard sequence number. If you do not have WINSTA_ACCESSCLIPBOARD access to the window station, the function returns zero.

Remarks

The system keeps a serial number for the clipboard for each window station. This number is incremented whenever the contents of the clipboard change or the clipboard is emptied. You can track this value to determine whether the clipboard contents have changed and optimize creating DataObjects. If clipboard rendering is delayed, the sequence number is not incremented until the changes are rendered.

Requirements

Windows NT/2000: Requires Windows 2000 or later.

Windows 95/98: Requires Windows 98 or later.

3.169 GetClipboardViewer

The GetClipboardViewer function retrieves the handle to the first window in the clipboard viewer chain.

```
GetClipboardViewer: procedure;
    @stdcall;
    @returns( "eax" );
    @external( "__imp__GetClipboardViewer@0" );
```

Parameters

This function has no parameters.

Return Values

If the function succeeds, the return value is the handle to the first window in the clipboard viewer chain.

If there is no clipboard viewer, the return value is NULL. To get extended error information, call GetLastError.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.170 GetComboBoxInfo

The GetComboBoxInfo function retrieves information about the specified combo box.

```
GetComboBoxInfo: procedure
(
        hwndCombo   :dword;
    var pcbi        :COMBOBOXINFO
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__GetComboBoxInfo@8" );
```

Parameters

hwndCombo

[in] Handle to the combo box.

pcbi

[out] Pointer to a COMBOBOXINFO structure that receives the information.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

The CB_GETCOMBOBOXINFO message is equivalent to this function.

Requirements

Windows NT/2000: Requires Windows NT 4.0 SP6 or later.

Windows 95/98: Requires Windows 98 or later.

3.171 GetCursor

The GetCursor function retrieves a handle to the current cursor.

```
GetCursor: procedure;
    @stdcall;
    @returns( "eax" );
    @external( "__imp__GetCursor@0" );
```

Parameters

This function has no parameters.

Return Values

The return value is the handle to the current cursor. If there is no cursor, the return value is NULL.

Windows 98 and Windows NT 4.0 SP3 and later: To get information on the global cursor, even if it is not owned by the current thread, use GetCursorInfo.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.172 GetCursorInfo

The GetCursorInfo function retrieves information about the global cursor.

```
GetCursorInfo: procedure
(
    var pci         :CURSORINFO
);
```

```
    @stdcall;
    @returns( "eax" );
    @external( "__imp__GetCursorInfo@4" );
```

Parameters

pci

[out] Pointer to a CURSORINFO structure that receives the information.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Requirements

Windows NT/2000: Requires Windows NT 4.0 SP6 or later.

Windows 95/98: Requires Windows 98 or later.

3.173 GetCursorPos

The GetCursorPos function retrieves the cursor's position, in screen coordinates.

```
GetCursorPos: procedure
(
    var lpPoint     :POINT
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__GetCursorPos@4" );
```

Parameters

lpPoint

[out] Pointer to a POINT structure that receives the screen coordinates of the cursor.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

The cursor position is always specified in screen coordinates and is not affected by the mapping mode of the window that contains the cursor.

The calling process must have WINSTA_READATTRIBUTES access to the window station.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.174 GetDC

The GetDC function retrieves a handle to a display device context (DC) for the client area of a specified window or for the entire screen. You can use the returned handle in subsequent GDI functions to draw in the DC.

The GetDCEx function is an extension to GetDC, which gives an application more control over how and whether clipping occurs in the client area.

```
GetDC: procedure
(
    hWnd            :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__GetDC@4" );
```

Parameters

hWnd

[in] Handle to the window whose DC is to be retrieved. If this value is NULL, GetDC retrieves the DC for the entire screen.

Windows 98, Windows 2000: To get the DC for a specific display monitor, use the EnumDisplayMonitors and CreateDC functions.

Return Values

If the function succeeds, the return value is a handle to the DC for the specified window's client area.

If the function fails, the return value is NULL.

Windows NT/2000: To get extended error information, call GetLastError.

Remarks

The GetDC function retrieves a common, class, or private DC depending on the class style specified for the specified window. For common DCs, GetDC assigns default attributes to the DC each time it is retrieved. For class and private DCs, GetDC leaves the previously assigned attributes unchanged.

Note that the handle to the DC can only be used by a single thread at any one time.

After painting with a common DC, the ReleaseDC function must be called to release the DC. Class and private DCs do not have to be released. ReleaseDC must be called from the same thread that called GetDC. The number of DCs is limited only by available memory.

Windows 95/98: There are only 5 common DCs available per thread, thus failure to release a DC can prevent other applications from accessing one.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.175 GetDCEx

The GetDCEx function retrieves a handle to a display device context (DC) for the client area of a specified window or for the entire screen. You can use the returned handle in subsequent GDI functions to draw in the DC.

This function is an extension to the GetDC function, which gives an application more control over how and

whether clipping occurs in the client area.

```
GetDCEx: procedure
(
    hWnd            :dword;
    hrgnClip        :dword;
    flags           :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__GetDCEx@12" );
```

Parameters

hWnd

[in] Handle to the window whose DC is to be retrieved. If this value is NULL, GetDCEx retrieves the DC for the entire screen.

Windows 98, Windows 2000: To get the DC for a specific display monitor, use the EnumDisplayMonitors and CreateDC functions.

hrgnClip

[in] Specifies a clipping region that may be combined with the visible region of the DC. If the value of flags is DCX_INTERSECTRGN or DCX_EXCLUDERGN, then the operating system assumes ownership of the region and will automatically delete it when it is no longer needed. In this case, applications should not use the region—not even delete it—after a successful call to GetDCEx.

flags

[in] Specifies how the DC is created. This parameter can be one or more of the following values.

Value	Meaning
DCX_WINDOW	Returns a DC that corresponds to the window rectangle rather than the client rectangle.
DCX_CACHE	Returns a DC from the cache, rather than the OWNDC or CLASSDC window. Essentially overrides CS_OWNDC and CS_CLASSDC.
DCX_PARENTCLIP	Uses the visible region of the parent window. The parent's WS_CLIPCHILDREN and CS_PARENTDC style bits are ignored. The origin is set to the upper-left corner of the window identified by *hWnd*.
DCX_CLIPSIBLINGS	Excludes the visible regions of all sibling windows above the window identified by hWnd.
DCX_CLIPCHILDREN	Excludes the visible regions of all child windows below the window identified by hWnd.
DCX_NORESETATTRS	Does not reset the attributes of this DC to the default attributes when this DC is released.
DCX_LOCKWINDOWUPDATE	Allows drawing even if there is a LockWindowUpdate call in effect that would otherwise exclude this window. Used for drawing during tracking.
DCX_EXCLUDERGN	The clipping region identified by hrgnClip is excluded from the visible region of the returned DC.
DCX_INTERSECTRGN	The clipping region identified by hrgnClip is intersected with the visible region of the returned DC.
DCX_VALIDATE	When specified with DCX_INTERSECTUPDATE, causes the DC to be completely validated. Using this function with both DCX_INTERSECTUPDATE and DCX_VALIDATE is identical to using the BeginPaint function.

Return Values

If the function succeeds, the return value is the handle to the DC for the specified window.

If the function fails, the return value is NULL. An invalid value for the hWnd parameter will cause the function to fail.

Windows NT/2000: To get extended error information, call GetLastError.

Remarks

Unless the display DC belongs to a window class, the ReleaseDC function must be called to release the DC after painting. Also, ReleaseDC must be called from the same thread that called GetDCEx. The number of DCs is limited only by available memory.

Windows 95/98: There are only five common DCs available at any time, thus failure to release a DC can prevent other applications from accessing one.

The function returns a handle to a DC that belongs to the window's class if CS_CLASSDC, CS_OWNDC or CS_PARENTDC was specified as a style in the WNDCLASS structure when the class was registered.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.176 GetDesktopWindow

The GetDesktopWindow function returns a handle to the desktop window. The desktop window covers the entire screen. The desktop window is the area on top of which all icons and other windows are painted.

```
GetDesktopWindow: procedure;
    @stdcall;
    @returns( "eax" );
    @external( "__imp__GetDesktopWindow@0" );
```

Parameters

This function has no parameters.

Return Values

The return value is a handle to the desktop window.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.177 GetDialogBaseUnits

The GetDialogBaseUnits function retrieves the system's dialog base units, which are the average width and height of characters in the system font. For dialog boxes that use the system font, you can use these values to convert between dialog template units, as specified in dialog box templates, and pixels. For dialog boxes that do not use the system font, the conversion from dialog template units to pixels depends on the font used by the dialog

box.

For either type of dialog box, it is easier to use the MapDialogRect function to perform the conversion. MapDialogRect takes the font into account and correctly converts a rectangle from dialog template units into pixels.

```
GetDialogBaseUnits: procedure;
    @stdcall;
    @returns( "eax" );
    @external( "__imp__GetDialogBaseUnits@0" );
```

Parameters

This function has no parameters.

Return Values

The function returns the dialog base units. The low-order word of the return value contains the horizontal dialog box base unit, and the high-order word contains the vertical dialog box base unit.

Remarks

The horizontal base unit returned by GetDialogBaseUnits is equal to the average width, in pixels, of the characters in the system font; the vertical base unit is equal to the height, in pixels, of the font.

For a dialog box that does not use the system font, the base units are the average width and height, in pixels, of the characters in the dialog's font. You can use the GetTextMetrics and GetTextExtentPoint32 functions to calculate these values for a selected font. However, by using the MapDialogRect function, you can avoid errors that might result if your calculations differ from those performed by the system.

Each horizontal base unit is equal to 4 horizontal dialog template units; each vertical base unit is equal to 8 vertical dialog template units. Therefore, to convert dialog template units to pixels, use the following formulas:

pixelX = MulDiv(templateunitX, baseunitX, 4);

pixelY = MulDiv(templateunitY, baseunitY, 8);

Similarly, to convert from pixels to dialog template units, use the following formulas:

templateunitX = MulDiv(pixelX, 4, baseunitX);

templateunitY = MulDiv(pixelY, 8, baseunitY);

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.178 GetDlgCtrlID

The GetDlgCtrlID function retrieves the identifier of the specified control.

```
GetDlgCtrlID: procedure
(
    hwndCtl         :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__GetDlgCtrlID@4" );
```

Parameters

hwndCtl

[in] Handle to the control.

Return Values

If the function succeeds, the return value is the identifier of the control.

If the function fails, the return value is zero. An invalid value for the hwndCtl parameter, for example, will cause the function to fail. To get extended error information, call GetLastError.

Remarks

GetDlgCtrlID accepts child window handles as well as handles of controls in dialog boxes. An application sets the identifier for a child window when it creates the window by assigning the identifier value to the hmenu parameter when calling the CreateWindow or CreateWindowEx function.

Although GetDlgCtrlID may return a value if hwndCtl is a handle to a top-level window, top-level windows cannot have identifiers and such a return value is never valid.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.179 GetDlgItem

The GetDlgItem function retrieves a handle to a control in the specified dialog box.

```
GetDlgItem: procedure
(
    hDlg            :dword;
    nIDDlgItem      :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__GetDlgItem@8" );
```

Parameters

hDlg

[in] Handle to the dialog box that contains the control.

nIDDlgItem

[in] Specifies the identifier of the control to be retrieved.

Return Values

If the function succeeds, the return value is the window handle of the specified control.

If the function fails, the return value is NULL, indicating an invalid dialog box handle or a nonexistent control. To get extended error information, call GetLastError.

Remarks

You can use the GetDlgItem function with any parent-child window pair, not just with dialog boxes. As long as the hDlg parameter specifies a parent window and the child window has a unique identifier (as specified by the hMenu parameter in the CreateWindow or CreateWindowEx function that created the child window), GetDlgItem returns a valid handle to the child window.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.180 GetDlgItemInt

The GetDlgItemInt function translates the text of a specified control in a dialog box into an integer value.

```
GetDlgItemInt: procedure
(
        hDlg            :dword;
        nIDDlgItem      :dword;
    var lpTranslated    :boolean;
        bSigned         :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__GetDlgItemInt@16" );
```

Parameters

hDlg

[in] Handle to the dialog box that contains the control of interest.

nIDDlgItem

[in] Specifies the identifier of the control whose text is to be translated.

lpTranslated

[out] Pointer to a variable that receives a success or failure value (TRUE indicates success, FALSE indicates failure).

If this parameter is NULL, the function returns no information about success or failure.

bSigned

[in] Specifies whether the function should examine the text for a minus sign at the beginning and return a signed integer value if it finds one (TRUE specifies this should be done, FALSE that it should not).

Return Values

If the function succeeds, the variable pointed to by lpTranslated is set to TRUE, and the return value is the translated value of the control text.

If the function fails, the variable pointed to by lpTranslated is set to FALSE, and the return value is zero. Note that, since zero is a possible translated value, a return value of zero does not by itself indicate failure.

If lpTranslated is NULL, the function returns no information about success or failure.

If the bSigned parameter is TRUE, specifying that the value to be retrieved is a signed integer value, cast the return value to an int type. To get extended error information, call GetLastError.

Remarks

The GetDlgItemInt function retrieves the text of the specified control by sending the control a WM_GETTEXT message. The function translates the retrieved text by stripping any extra spaces at the beginning of the text and then converting the decimal digits. The function stops translating when it reaches the end of the text or encounters a nonnumeric character.

If the bSigned parameter is TRUE, the GetDlgItemInt function checks for a minus sign (–) at the beginning of the text and translates the text into a signed integer value. Otherwise, the function creates an unsigned integer value.

The GetDlgItemInt function returns zero if the translated value is greater than INT_MAX (for signed numbers) or UINT_MAX (for unsigned numbers).

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.181 GetDlgItemText

The GetDlgItemText function retrieves the title or text associated with a control in a dialog box.

```
GetDlgItemText: procedure
(
    hDlg            :dword;
    nIDDlgItem      :dword;
    lpString        :string;
    nMaxCount       :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__GetDlgItemTextA@16" );
```

Parameters

hDlg

[in] Handle to the dialog box that contains the control.

nIDDlgItem

[in] Specifies the identifier of the control whose title or text is to be retrieved.

lpString

[out] Pointer to the buffer to receive the title or text.

nMaxCount

[in] Specifies the maximum length, in TCHARs, of the string to be copied to the buffer pointed to by lpString. If the length of the string exceeds the limit, the string is truncated.

Return Values

If the function succeeds, the return value specifies the number of TCHARs copied to the buffer, not including the terminating null character.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

The GetDlgItemText function sends a WM_GETTEXT message to the control.

For the ANSI version of the function, the number of TCHARs is the number of bytes; for the Unicode version, it is the number of characters.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.182 GetDoubleClickTime

The GetDoubleClickTime function retrieves the current double-click time for the mouse. A double-click is a series of two clicks of the mouse button, the second occurring within a specified time after the first. The double-click time is the maximum number of milliseconds that may occur between the first and second click of a double-click.

```
GetDoubleClickTime: procedure;
    @stdcall;
    @returns( "eax" );
    @external( "__imp__GetDoubleClickTime@0" );
```

Parameters

This function has no parameters.

Return Values

The return value specifies the current double-click time, in milliseconds.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.183 GetFocus

The GetFocus function retrieves the handle to the window that has the keyboard focus, if the window is attached to the calling thread's message queue.

```
GetFocus: procedure;
    @stdcall;
    @returns( "eax" );
    @external( "__imp__GetFocus@0" );
```

Parameters

This function has no parameters.

Return Values

The return value is the handle to the window with the keyboard focus. If the calling thread's message queue does not have an associated window with the keyboard focus, the return value is NULL.

Remarks

GetFocus returns the window with the keyboard focus for the current thread's message queue. If GetFocus returns NULL, another thread's queue may be attached to a window that has the keyboard focus.

Use the GetForegroundWindow function to retrieve the handle to the window with which the user is currently working. You can associate your thread's message queue with the windows owned by another thread by using the AttachThreadInput function.

Windows 98 and Windows NT 4.0 SP3 and later: To get the window with the keyboard focus on the foreground

queue or the queue of another thread, use the GetGUIThreadInfo function.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.184 GetForegroundWindow

The GetForegroundWindow function returns a handle to the foreground window (the window with which the user is currently working). The system assigns a slightly higher priority to the thread that creates the foreground window than it does to other threads.

```
GetForegroundWindow: procedure;
    @stdcall;
    @returns( "eax" );
    @external( "__imp__GetForegroundWindow@0" );
```

Parameters

This function has no parameters.

Return Values

The return value is a handle to the foreground window. The foreground window can be NULL in certain circumstances, such as when a window is losing activation.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.185 GetGUIThreadInfo

The GetGUIThreadInfo function retrieves information about the active window or a specified graphical user interface (GUI) thread.

```
GetGUIThreadInfo: procedure
(
        idThread    :dword;
    var lpgui       :GUITHREADINFO
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__GetGUIThreadInfo@8" );
```

Parameters

idThread

[in] Identifies the thread for which information is to be retrieved. To retrieve this value, use the GetWindowThreadProcessId function. If this parameter is NULL, the function returns information for the foreground thread.

lpgui

[out] Pointer to a GUITHREADINFO structure that receives information describing the thread.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

This function succeeds even if the active window is not owned by the calling process. If the specified thread does not exist or have an input queue, the function will fail.

This function is useful for retrieving out-of-context information about a thread. The information retrieved is the same as if an application retrieved the information about itself.

Requirements

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.

Windows 95/98: Requires Windows 98 or later.

3.186 GetGuiResources

The GetGuiResources function retrieves the count of handles to graphical user interface (GUI) objects in use by the specified process.

```
GetGuiResources: procedure
(
    hProcess        :dword;
    uiFlags         :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__GetGuiResources@8" );
```

Parameters

hProcess

[in] Handle to the process. The handle must have the PROCESS_QUERY_INFORMATION access right. For more information, see Process Security and Access Rights.

uiFlags

[in] Specifies the GUI object type. This parameter can be one of the following values.

Value	Meaning
GR_GDIOBJECTS	Return the count of GDI objects.
GR_USEROBJECTS	Return the count of USER objects.

Return Values

If the function succeeds, the return value is the count of handles to GUI objects in use by the process. If no GUI objects are in use, the return value is zero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

A process without a graphical user interface does not use GUI resources, therefore, GetGuiResources will return

zero.

Requirements

Windows NT/2000: Requires Windows 2000 or later.

Windows 95/98: Unsupported.

3.187 GetIconInfo

The GetIconInfo function retrieves information about the specified icon or cursor.

```
GetIconInfo: procedure
(
        hIcon       :dword;
    var piconinfo   :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__GetIconInfo@8" );
```

Parameters

hIcon

[in] Handle to the icon or cursor. To retrieve information about a standard icon or cursor, specify one of the following values.

Value	Meaning
IDC_APPSTARTING	Standard arrow and small hourglass cursor.
IDC_ARROW	Standard arrow cursor.
IDC_CROSS	Crosshair cursor.
IDC_HAND	Windows 2000: Hand cursor.
IDC_HELP	Arrow and question mark cursor.
IDC_IBEAM	I-beam cursor.
IDC_NO	Slashed circle cursor.
IDC_SIZEALL	Four-pointed arrow cursor pointing north, south, east, and west.
IDC_SIZENESW	Double-pointed arrow cursor pointing northeast and southwest.
IDC_SIZENS	Double-pointed arrow cursor pointing north and south.
IDC_SIZENWSE	Double-pointed arrow cursor pointing northwest and southeast.
IDC_SIZEWE	Double-pointed arrow cursor pointing west and east.
IDC_UPARROW	Vertical arrow cursor.
IDC_WAIT	Hourglass cursor.
IDI_APPLICATION	Application icon.
IDI_ASTERISK	Asterisk icon.
IDI_EXCLAMATION	Exclamation point icon.
IDI_HAND	Stop sign icon.
IDI_QUESTION	Question-mark icon.
IDI_WINLOGO	Windows logo icon.

piconinfo

[out] Pointer to an ICONINFO structure. The function fills in the structure's members.

Return Values

If the function succeeds, the return value is nonzero and the function fills in the members of the specified ICONINFO structure.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

GetIconInfo creates bitmaps for the hbmMask and hbmColor members of ICONINFO. The calling application must manage these bitmaps and delete them when they are no longer necessary.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.188 GetInputState

The GetInputState function determines whether there are mouse-button or keyboard messages in the calling thread's message queue.

```
GetInputState: procedure;
    @stdcall;
    @returns( "eax" );
    @external( "__imp__GetInputState@0" );
```

Parameters

This function has no parameters.

Return Values

If the queue contains one or more new mouse-button or keyboard messages, the return value is nonzero.

If the there are no new mouse-button or keyboard messages in the queue, the return value is zero.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.189 GetKBCodePage

The GetKBCodePage function returns the current code page.

Note This function is provided only for compatibility with 16-bit versions of Windows. Win32-based applications should use the GetOEMCP function to retrieve the OEM code-page identifier for the system.

```
GetKBCodePage: procedure;
    @stdcall;
    @returns( "eax" );
    @external( "__imp__GetKBCodePage@0" );
```

Parameters

This function has no parameters.

Return Values

The return value is an OEM code-page identifier, or it is the default identifier if the registry value is not readable. For a list of OEM code-page identifiers, see GetOEMCP.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.190 GetKeyNameText

The GetKeyNameText function retrieves a string that represents the name of a key.

```
GetKeyNameText: procedure
(
    _lParam         :dword;
    lpString        :string;
    nSize           :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__GetKeyNameTextA@12" );
```

Parameters

lParam

[in] Specifies the second parameter of the keyboard message (such as WM_KEYDOWN) to be processed. The function interprets the following portions of lParam.

Bits	Meaning
16–23	Scan code.
24	Extended-key flag. Distinguishes some keys on an enhanced keyboard.
25	"Don't care" bit. The application calling this function sets this bit to indicate that the function should not distinguish between left and right CTRL and SHIFT keys, for example.

lpString

[out] Pointer to a buffer that will receive the key name.

nSize

[in] Specifies the maximum length, in TCHARs, of the key name, including the terminating null character. (This parameter should be equal to the size of the buffer pointed to by the lpString parameter.)

Return Values

If the function succeeds, a null-terminated string is copied into the specified buffer, and the return value is the length of the string, in TCHARs, not counting the terminating null character.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

The format of the key-name string depends on the current keyboard layout. The keyboard driver maintains a list of names in the form of character strings for keys with names longer than a single character. The key name is translated according to the layout of the currently installed keyboard. The name of a character key is the character itself. The names of dead keys are spelled out in full.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.191 GetKeyState

The GetKeyState function retrieves the status of the specified virtual key. The status specifies whether the key is up, down, or toggled (on, off—alternating each time the key is pressed).

```
GetKeyState: procedure
(
    nVirtKey        :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__GetKeyState@4" );
```

Parameters

nVirtKey

[in] Specifies a virtual key. If the desired virtual key is a letter or digit (A through Z, a through z, or 0 through 9), nVirtKey must be set to the ASCII value of that character. For other keys, it must be a virtual-key code.

If a non-English keyboard layout is used, virtual keys with values in the range ASCII A through Z and 0 through 9 are used to specify most of the character keys. For example, for the German keyboard layout, the virtual key of value ASCII O (0x4F) refers to the "o" key, whereas VK_OEM_1 refers to the "o with umlaut" key.

Return Values

The return value specifies the status of the specified virtual key, as follows:

- If the high-order bit is 1, the key is down; otherwise, it is up.
- If the low-order bit is 1, the key is toggled. A key, such as the CAPS LOCK key, is toggled if it is turned on. The key is off and untoggled if the low-order bit is 0. A toggle key's indicator light (if any) on the keyboard will be on when the key is toggled, and off when the key is untoggled.

Remarks

The key status returned from this function changes as a thread reads key messages from its message queue. The status does not reflect the interrupt-level state associated with the hardware. Use the GetAsyncKeyState function to retrieve that information.

An application calls GetKeyState in response to a keyboard-input message. This function retrieves the state of the key when the input message was generated.

To retrieve state information for all the virtual keys, use the GetKeyboardState function.

An application can use the virtual-key code constants VK_SHIFT, VK_CONTROL, and VK_MENU as values for the nVirtKey parameter. This gives the status of the SHIFT, CTRL, or ALT keys without distinguishing between left and right. An application can also use the following virtual-key code constants as values for nVirtKey to distinguish between the left and right instances of those keys.

VK_LSHIFT

VK_RSHIFT

VK_LCONTROL

VK_RCONTROL

VK_LMENU

VK_RMENU

These left- and right-distinguishing constants are available to an application only through the GetKeyboardState, SetKeyboardState, GetAsyncKeyState, GetKeyState, and MapVirtualKey functions.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.192 GetKeyboardLayout

The GetKeyboardLayout function retrieves the active input locale identifier (formerly called the keyboard layout) for the specified thread. If the idThread parameter is zero, the input locale identifier for the active thread is returned.

```
GetKeyboardLayout: procedure
(
    idThread        :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__GetKeyboardLayout@4" );
```

Parameters

idThread

[in] Identifies the thread to query or is zero for the current thread.

Return Values

The return value is the input locale identifier for the thread. The low word contains a language identifier for the input language and the high word contains a device handle for the physical layout of the keyboard.

Remarks

The input locale identifier is a broader concept than a keyboard layout, since it can also encompass a speech-to-text converter, an IME, or any other form of input.

Since the keyboard layout can be dynamically changed, applications that cache information about the current keyboard layout should process the WM_INPUTLANGCHANGE message to be informed of changes in the input language.

Requirements

Windows NT/2000: Requires Windows NT 4.0 or later.

Windows 95/98: Requires Windows 95 or later.

3.193 GetKeyboardLayoutList

The GetKeyboardLayoutList function retrieves the input locale identifiers (formerly called keyboard layout handles) corresponding to the current set of input locales in the system. The function copies the identifiers to the specified buffer.

```
GetKeyboardLayoutList: procedure
(
        nBuff       :dword;
    var lpList      :var
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__GetKeyboardLayoutList@8" );
```

Parameters

nBuff

[in] Specifies the maximum number of handles that the buffer can hold.

lpList

[out] Pointer to the buffer that receives the array of input locale identifiers.

Return Values

If the function succeeds, the return value is the number of input locale identifiers copied to the buffer or, if nBuff is zero, the return value is the size, in array elements, of the buffer needed to receive all current input locale identifiers.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

The input locale identifier is a broader concept than a keyboard layout, since it can also encompass a speech-to-text converter, an IME, or any other form of input.

Requirements

Windows NT/2000: Requires Windows NT 4.0 or later.

Windows 95/98: Requires Windows 95 or later.

3.194 GetKeyboardLayoutName

The GetKeyboardLayoutName function retrieves the name of the active input locale identifier (formerly called the keyboard layout).

```
GetKeyboardLayoutName: procedure
(
    pwszKLID        :string
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__GetKeyboardLayoutNameA@4" );
```

Parameters

pwszKLID

[out] Pointer to the buffer (of at least KL_NAMELENGTH characters in length) that receives the name of the input locale identifier, including the NULL terminator. This will be a copy of the string provided to the LoadKeyboardLayout function, unless layout substitution took place.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

The input locale identifier is a broader concept than a keyboard layout, since it can also encompass a speech-to-text converter, an IME, or any other form of input.

Windows NT/2000: GetKeyboardLayoutName receives the name of the active input locale identifier for the system.

Windows 95: GetKeyboardLayoutName receives the name of the active input locale identifier for the calling thread.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.195 GetKeyboardState

The GetKeyboardState function copies the status of the 256 virtual keys to the specified buffer.

```
GetKeyboardState: procedure
(
    var lpKeyState  :var
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__GetKeyboardState@4" );
```

Parameters

lpKeyState

[in] Pointer to the 256-byte array that will receive the status data for each virtual key.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

An application can call this function to retrieve the current status of all the virtual keys. The status changes as a thread removes keyboard messages from its message queue. The status does not change as keyboard messages are posted to the thread's message queue, nor does it change as keyboard messages are posted to or retrieved from message queues of other threads. (Exception: Threads that are connected through AttachThreadInput share the same keyboard state.)

When the function returns, each member of the array pointed to by the lpKeyState parameter contains status data for a virtual key. If the high-order bit is 1, the key is down; otherwise, it is up. If the low-order bit is 1, the key is toggled. A key, such as the CAPS LOCK key, is toggled if it is turned on. The key is off and untoggled if the low-order bit is 0. A toggle key's indicator light (if any) on the keyboard will be on when the key is toggled, and off when the key is untoggled.

To retrieve status information for an individual key, use the GetKeyState function. To retrieve the current state for

an individual key regardless of whether the corresponding keyboard message has been retrieved from the message queue, use the GetAsyncKeyState function.

An application can use the virtual-key code constants VK_SHIFT, VK_CONTROL and VK_MENU as indices into the array pointed to by lpKeyState. This gives the status of the SHIFT, CTRL, or ALT keys without distinguishing between left and right. An application can also use the following virtual-key code constants as indices to distinguish between the left and right instances of those keys:

VK_LSHIFT

VK_RSHIFT

VK_LCONTROL

VK_RCONTROL

VK_LMENU

VK_RMENU

These left- and right-distinguishing constants are available to an application only through the GetKeyboardState, SetKeyboardState, GetAsyncKeyState, GetKeyState, and MapVirtualKey functions.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.196 GetKeyboardType

The GetKeyboardType function retrieves information about the current keyboard.

```
GetKeyboardType: procedure
(
    nTypeFlag       :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__GetKeyboardType@4" );
```

Parameters

nTypeFlag

[in] Specifies the type of keyboard information to be retrieved. This parameter can be one of the following values.

Value	Meaning
0	Keyboard type
1	Keyboard subtype
2	Number of function keys on the keyboard

Return Values

If the function succeeds, the return value specifies the requested information.

If the function fails and nTypeFlag is not one, the return value is zero; zero is a valid return value when nType-Flag is one (keyboard subtype). To get extended error information, call GetLastError.

Remarks

The type may be one of the following values.

Value	Meaning
1	IBM PC/XT or compatible (83-key) keyboard
2	Olivetti "ICO" (102-key) keyboard
3	IBM PC/AT (84-key) or similar keyboard
4	IBM enhanced (101- or 102-key) keyboard
5	Nokia 1050 and similar keyboards
6	Nokia 9140 and similar keyboards
7	Japanese keyboard

The subtype is an original equipment manufacturer (OEM)-dependent value.

The application can also determine the number of function keys on a keyboard from the keyboard type. Follow-ing are the number of function keys for each keyboard type.

Type	Number of function keys
1	10
2	12 (sometimes 18)
3	10
4	12
5	10
6	24
7	Hardware dependent and specified by the OEM

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.197 GetLastActivePopup

The GetLastActivePopup function determines which pop-up window owned by the specified window was most recently active.

```
GetLastActivePopup: procedure
(
    hWnd            :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__GetLastActivePopup@4" );
```

Parameters

hWnd

[in] Handle to the owner window.

Return Values

The return value identifies the most recently active pop-up window. The return value is the same as the hWnd parameter, if any of the following conditions are met:

- The window identified by hWnd was most recently active.
- The window identified by hWnd does not own any pop-up windows.
- The window identified by hWnd is not a top-level window or it is owned by another window.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.198 GetLastInputInfo

The GetLastInputInfo function retrieves the time of the last input event.

```
GetLastInputInfo: procedure
(
    var plii    :LASTINPUTINFO
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__GetLastInputInfo@4" );
```

Parameters

plii

[out] Pointer to a LASTINPUTINFO structure that receives the time of the last input event.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Remarks

This is useful for input idle detection.

Requirements

Windows NT/2000: Requires Windows 2000 or later.

Windows 95/98: Unsupported.

3.199 GetListBoxInfo

The GetListBoxInfo function retrieves information about the specified list box.

```
GetListBoxInfo: procedure
(
    hwnd            :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__GetListBoxInfo@4" );
```

Parameters

hwnd

[in] Handle to the list box whose information is to be retrieved.

Return Values

The return value is the number of items per column.

Requirements

Windows NT/2000: Requires Windows NT 4.0 SP6 or later.

Windows 95/98: Requires Windows 98 or later.

3.200 GetMenu

The GetMenu function retrieves a handle to the menu assigned to the specified window.

```
GetMenu: procedure
(
    hWnd            :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__GetMenu@4" );
```

Parameters

hWnd

[in] Handle to the window whose menu handle is to be retrieved.

Return Values

The return value is a handle to the menu. If the specified window has no menu, the return value is NULL. If the window is a child window, the return value is undefined.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.201 GetMenuBarInfo

The GetMenuBarInfo function retrieves information about the specified menu bar.

```
GetMenuBarInfo: procedure
(
        hwnd        :dword;
        idObject    :dword;
        idItem      :dword;
    var pmbi        :MENUBARINFO
);
    @stdcall;
```

```
    @returns( "eax" );
    @external( "__imp__GetMenuBarInfo@16" );
```

Parameters

hwnd

[in] Handle to the window (menu bar) whose information is to be retrieved.

idObject

[in] Specifies the menu object. This parameter can be one of the following values.

Value	Meaning
OBJID_CLIENT	The popup menu associated with the window.
OBJID_MENU	The menu bar associated with the window (see the GetMenu function).
OBJID_SYSMENU	The system menu associated with the window (see the GetSystemMenu function).

idItem

[in] Specifies the item for which to retrieve information. If this parameter is zero, the function retrieves information about the menu itself. If this parameter is 1, the function retrieves information about the first item on the menu, and so on.

pmbi

[out] Pointer to a MENUBARINFO structure that receives the information.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Requirements

Windows NT/2000: Requires Windows NT 4.0 SP6 or later.

Windows 95/98: Requires Windows 98 or later.

3.202 GetMenuCheckMarkDimensions

The GetMenuCheckMarkDimensions function returns the dimensions of the default check-mark bitmap. The system displays this bitmap next to selected menu items. Before calling the SetMenuItemBitmaps function to replace the default check-mark bitmap for a menu item, an application must determine the correct bitmap size by calling GetMenuCheckMarkDimensions.

Note The GetMenuCheckMarkDimensions function is included only for compatibility with 16-bit versions of Windows. For Win32-based applications, use the GetSystemMetrics function with the CXMENUCHECK and CYMENUCHECK values to retrieve the bitmap dimensions.

```
GetMenuCheckMarkDimensions: procedure;
    @stdcall;
    @returns( "eax" );
    @external( "__imp__GetMenuCheckMarkDimensions@0" );
```

Parameters

This function has no parameters.

Return Values

The return value specifies the height and width, in pixels, of the default check-mark bitmap. The high-order word contains the height; the low-order word contains the width.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.203 GetMenuContextHelpId

Retrieves the Help context identifier associated with the specified menu.

```
GetMenuContextHelpId: procedure
(
    hMenu           :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__GetMenuContextHelpId@4" );
```

Parameters

hmenu

Handle to the menu for which the Help context identifier is to be retrieved.

Return Values

Returns the Help context identifier if the menu has one, or zero otherwise.

See Also

SetMenuContextHelpId

Requirements

Windows NT/2000: Requires Windows NT 3.51 or later.

Windows 95/98: Requires Windows 95 or later.

3.204 GetMenuDefaultItem

The GetMenuDefaultItem function determines the default menu item on the specified menu.

```
GetMenuDefaultItem: procedure
(
    hMenu           :dword;
    fByPos          :boolean;
    gmdiFlags       :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__GetMenuDefaultItem@12" );
```

Parameters

hMenu

[in] Handle to the menu for which to retrieve the default menu item.

fByPos

[in] Specifies whether to retrieve the menu item's identifier or its position. If this parameter is FALSE, the identifier is returned. Otherwise, the position is returned.

gmdiFlags

[in] Specifies how the function searches for menu items. This parameter can be zero or more of the following values.

Value	Meaning
GMDI_GOINTOPOPUPS	Specifies that if the default item is one that opens a submenu, the function is to search recursively in the corresponding submenu. If the submenu has no default item, the return value identifies the item that opens the submenu. By default, the function returns the first default item on the specified menu, regardless of whether it is an item that opens a submenu.
GMDI_USEDISABLED	Specifies that the function is to return a default item, even if it is disabled. By default, the function skips disabled or grayed items.

Return Values

If the function succeeds, the return value is the identifier or position of the menu item.

If the function fails, the return value is – 1. To get extended error information, call GetLastError.

Requirements

Windows NT/2000: Requires Windows NT 4.0 or later.

Windows 95/98: Requires Windows 95 or later.

3.205 GetMenuInfo

The GetMenuInfo function gets information about a specified menu.

```
GetMenuInfo: procedure
(
        hmenu       :dword;
    var lpcmi       :CMENUINFO
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__GetMenuInfo@8" );
```

Parameters

hmenu

[in] Handle for a menu.

lpcmi

[out] Pointer to a MENUINFO structure containing information for the menu.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Requirements

Windows NT/2000: Requires Windows 2000 or later.

Windows 95/98: Requires Windows 98 or later.

3.206 GetMenuItemCount

The GetMenuItemCount function determines the number of items in the specified menu.

```
GetMenuItemCount: procedure
(
    hMenu           :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__GetMenuItemCount@4" );
```

Parameters

hMenu

[in] Handle to the menu to be examined.

Return Values

If the function succeeds, the return value specifies the number of items in the menu.

If the function fails, the return value is –1. To get extended error information, call GetLastError.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.207 GetMenuItemID

The GetMenuItemID function retrieves the menu item identifier of a menu item located at the specified position in a menu.

```
GetMenuItemID: procedure
(
    hMenu           :dword;
    nPos            :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__GetMenuItemID@8" );
```

Parameters

hMenu

[in] Handle to the menu that contains the item whose identifier is to be retrieved.

nPos

[in] Specifies the zero-based relative position of the menu item whose identifier is to be retrieved.

Return Values

The return value is the identifier of the specified menu item. If the menu item identifier is NULL or if the specified item opens a submenu, the return value is -1.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.208 GetMenuItemInfo

The GetMenuItemInfo function retrieves information about a menu item.

```
GetMenuItemInfo: procedure
(
        hMenu       :dword;
        uItem       :dword;
        fByPosition :boolean;
    var lpmii       :MENUITEMINFO
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__GetMenuItemInfoA@16" );
```

Parameters

hMenu

[in] Handle to the menu that contains the menu item.

uItem

[in] Identifier or position of the menu item to get information about. The meaning of this parameter depends on the value of fByPosition.

fByPosition

[in] Specifies the meaning of uItem. If this parameter is FALSE, uItem is a menu item identifier. Otherwise, it is a menu item position.

lpmii

[in/out] Pointer to a MENUITEMINFO structure that specifies the information to retrieve and receives information about the menu item.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, use the GetLastError function.

Remarks

To retrieve a menu item of type MFT_STRING, first find the size of the string by setting the dwTypeData member of MENUITEMINFO to NULL and then calling GetMenuItemInfo. The value of cch is the size needed. Then allocate a buffer of this size, place the pointer to the buffer in dwTypeData, and call GetMenuItemInfo once again to fill the buffer with the string.

If the retrieved menu item is of some other type, then GetMenuItemInfo sets the dwTypeData member to a value whose type is specified by the fType member and sets cch to 0.

Windows 2000 and Windows 98: dwTypeData and cch are used with MIIM_STRING.

Requirements

Windows NT/2000: Requires Windows NT 4.0 or later.

Windows 95/98: Requires Windows 95 or later.

3.209 GetMenuItemRect

The GetMenuItemRect function retrieves the bounding rectangle for the specified menu item.

```
GetMenuItemRect: procedure
(
        hWnd        :dword;
        hMenu       :dword;
        uItem       :dword;
    var lprcItem    :RECT
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__GetMenuItemRect@16" );
```

Parameters

hWnd

[in] Handle to the window containing the menu.

Windows 98 and Windows 2000: If this value is NULL and the hMenu parameter represents a popup menu, the function will find the menu window.

hMenu

[in] Handle to a menu.

uItem

[in] Zero-based position of the menu item.

lprcItem

[out] Pointer to a RECT structure that receives the bounding rectangle of the specified menu item expressed in screen coordinates.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, use the GetLastError function.

Requirements

Windows NT/2000: Requires Windows NT 4.0 or later.

Windows 95/98: Requires Windows 95 or later.

3.210 GetMenuState

The GetMenuState function retrieves the menu flags associated with the specified menu item. If the menu item opens a submenu, this function also returns the number of items in the submenu.

Note The GetMenuState function has been superseded by the GetMenuItemInfo function. You can still use GetMenuState, however, if you do not need any of the extended features of GetMenuItemInfo.

```
GetMenuState: procedure
(
    hMenu       :dword;
    uId         :dword;
    uFlags      :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__GetMenuState@12" );
```

Parameters

hMenu

[in] Handle to the menu that contains the menu item whose flags are to be retrieved.

uId

[in] Specifies the menu item for which the menu flags are to be retrieved, as determined by the uFlags parameter.

uFlags

[in] Specifies how the uId parameter is interpreted. This parameter can be one of the following values.

Value	Description
MF_BYCOMMAND	Indicates that the uId parameter gives the identifier of the menu item. The MF_BYCOMMAND flag is the default if neither the MF_BYCOMMAND nor MF_BYPOSITION flag is specified.
MF_BYPOSITION	Indicates that the uId parameter gives the zero-based relative position of the menu item.

Return Values

If the specified item does not exist, the return value is -1.

If the menu item opens a submenu, the low-order byte of the return value contains the menu flags associated with the item, and the high-order byte contains the number of items in the submenu opened by the item.

Otherwise, the return value is a mask (Boolean OR) of the menu flags. Following are the menu flags associated with the menu item.

Value	Meaning
MF_CHECKED	A check mark is placed next to the item (for drop-down menus, submenus, and shortcut menus only).
MF_DISABLED	The item is disabled.
MF_GRAYED	The item is disabled and grayed.
MF_HILITE	The item is highlighted.

MF_MENUBARBREAK	This is the same as the MF_MENUBREAK flag, except for drop-down menus, submenus, and shortcut menus, where the new column is separated from the old column by a vertical line.
MF_MENUBREAK	The item is placed on a new line (for menu bars) or in a new column (for drop-down menus, submenus, and shortcut menus) without separating columns.
MF_OWNERDRAW	The item is owner-drawn.
MF_POPUP	Menu item is a submenu.
MF_SEPARATOR	There is a horizontal dividing line (for drop-down menus, sub-menus, and shortcut menus only).

Remarks

In addition, it is possible to test an item for a flag value of MF_ENABLED, MF_STRING, MF_UNCHECKED, or MF_UNHILITE. However, since these values equate to zero you must use an expression to test for them.

Flag	Expression to test for the flag
MF_ENABLED	! (Flag~(MF_DISABLED | MF_GRAYED
MF_STRING	! (Flag~(MF_BITMAP | MF_OWNERDRAW)
MF_UNCHECKED	! (Flag~MF_CHECKED)
MF_UNHILITE	! (Flag~HILITE)

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.211 GetMenuString

The GetMenuString function copies the text string of the specified menu item into the specified buffer.

Note The GetMenuString function has been superseded. Use the GetMenuItemInfo function to retrieve the menu item text.

```
GetMenuString: procedure
(
        hMenu       :dword;
        uIDItem     :dword;
    var lpString    :var;
        nMaxCount   :dword;
        uFlag       :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__GetMenuStringA@20" );
```

Parameters

hMenu

[in] Handle to the menu.

uIDItem

[in] Specifies the menu item to be changed, as determined by the uFlag parameter.

lpString

[out] Pointer to the buffer that receives the null-terminated string.

If lpString is NULL, the function returns the length of the menu string.

nMaxCount

[in] Specifies the maximum length, in characters, of the string to be copied. If the string is longer than the maximum specified in the nMaxCount parameter, the extra characters are truncated.

If nMaxCount is 0, the function returns the length of the menu string.

uFlag

[in] Specifies how the uIDItem parameter is interpreted. This parameter must be one of the following values.

Value	Meaning
MF_BYCOMMAND	Indicates that uIDItem gives the identifier of the menu item. If neither the MF_BYCOMMAND nor MF_BYPOSITION flag is specified, the MF_BYCOMMAND flag is the default flag.
MF_BYPOSITION	Indicates that uIDItem gives the zero-based relative position of the menu item.

Return Values

If the function succeeds, the return value specifies the number of characters copied to the buffer, not including the terminating null character.

If the function fails, the return value is zero.

If the specified item is not of type MFT_STRING, then the return value is zero.

Windows 98 and Windows 2000: MIIM_STRING supersedes MFT_STRING.

Remarks

The nMaxCount parameter must be one larger than the number of characters in the text string to accommodate the terminating null character.

If nMaxCount is 0, the function returns the length of the menu string.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.212 GetMessage

The GetMessage function retrieves a message from the calling thread's message queue. The function dispatches incoming sent messages until a posted message is available for retrieval.

Unlike GetMessage, the PeekMessage function does not wait for a message to be posted before returning.

```
GetMessage: procedure
(
    var lpMsg           :MSG;
        hWnd            :dword;
        wMsgFilterMin   :dword;
        wMsgFilterMax   :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__GetMessageA@16" );
```

Parameters

lpMsg

[out] Pointer to an MSG structure that receives message information from the thread's message queue.

hWnd

[in] Handle to the window whose messages are to be retrieved. The window must belong to the calling thread. The following value has a special meaning.

Value	Meaning
NULL	retrieves messages for any window that belongs to the calling thread and thread messages posted to the calling thread using the PostThreadMessage function.

wMsgFilterMin

[in] Specifies the integer value of the lowest message value to be retrieved. Use WM_KEYFIRST to specify the first keyboard message or WM_MOUSEFIRST to specify the first mouse message.

If wMsgFilterMin and wMsgFilterMax are both zero, GetMessage returns all available messages (that is, no range filtering is performed).

wMsgFilterMax

[in] Specifies the integer value of the highest message value to be retrieved. Use WM_KEYLAST to specify the first keyboard message or WM_MOUSELAST to specify the last mouse message.

If wMsgFilterMin and wMsgFilterMax are both zero, GetMessage returns all available messages (that is, no range filtering is performed).

Return Values

If the function retrieves a message other than WM_QUIT, the return value is nonzero.

If the function retrieves the WM_QUIT message, the return value is zero.

If there is an error, the return value is -1. For example, the function fails if hWnd is an invalid window handle or lpMsg is an invalid pointer. To get extended error information, call GetLastError.

Warning Because the return value can be nonzero, zero, or -1, avoid code like this:

while (GetMessage(lpMsg, hWnd, 0, 0)) ...

The possibility of a -1 return value means that such code can lead to fatal application errors. Instead, use code like this:

```
BOOL bRet;

while( (bRet = GetMessage( &msg, NULL, 0, 0 )) != 0)
{
    if (bRet == -1)
    {
        // handle the error and possibly exit
    }
    else
    {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
}
```

Remarks

An application typically uses the return value to determine whether to end the main message loop and exit the program.

The GetMessage function retrieves messages associated with the window identified by the hWnd parameter or any of its children, as specified by the IsChild function, and within the range of message values given by the wMsgFilterMin and wMsgFilterMax parameters. Note that an application can only use the low word in the wMsgFilterMin and wMsgFilterMax parameters; the high word is reserved for the system.

Note that GetMessage always retrieves WM_QUIT messages, no matter which values you specify for wMsgFilterMin and wMsgFilterMax.

During this call, the system delivers pending messages that were sent to windows owned by the calling thread using the SendMessage, SendMessageCallback, SendMessageTimeout, or SendNotifyMessage function. The system may also process internal events. Messages are processed in the following order:

- Sent messages
- Posted messages
- Input (hardware) messages and system internal events
- Sent messages (again)
- WM_PAINT messages
- WM_TIMER messages

To retrieve input messages before posted messages, use the wMsgFilterMin and wMsgFilterMax parameters.

GetMessage does not remove WM_PAINT messages from the queue. The messages remain in the queue until processed.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.213 GetMessageExtraInfo

The GetMessageExtraInfo function retrieves the extra message information for the current thread. Extra message information is an application- or driver-defined value associated with the current thread's message queue.

```
GetMessageExtraInfo: procedure;
    @stdcall;
    @returns( "eax" );
    @external( "__imp__GetMessageExtraInfo@0" );
```

Parameters

This function has no parameters.

Return Values

The return value specifies the extra information. The meaning of the extra information is device specific.

Remarks

To set a thread's extra message information, use the SetMessageExtraInfo function.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.214 GetMessagePos

The GetMessagePos function retrieves the cursor position for the last message retrieved by the GetMessage function.

To determine the current position of the cursor, use the GetCursorPos function.

```
GetMessagePos: procedure;
    @stdcall;
    @returns( "eax" );
    @external( "__imp__GetMessagePos@0" );
```

Parameters

This function has no parameters.

Return Values

The return value specifies the x- and y-coordinates of the cursor position. The x-coordinate is the low order int and the y-coordinate is the high-order int.

Remarks

As noted above, the x-coordinate is in the low-order int of the return value; the y-coordinate is in the high-order int (both represent signed values because they can take negative values on systems with multiple monitors). If the return value is assigned to a variable, you can use the MAKEPOINTS macro to obtain a POINTS structure from the return value. You can also use the GET_X_LPARAM or GET_Y_LPARAM macro to extract the x- or y-coordinate.

Important Do not use the LOWORD or HIWORD macros to extract the x- and y- coordinates of the cursor position because these macros return incorrect results on systems with multiple monitors. Systems with multiple monitor systems can have negative x- and y- coordinates, and LOWORD and HIWORD treat the coordinates as unsigned quantities.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.215 GetMessageTime

The GetMessageTime function retrieves the message time for the last message retrieved by the GetMessage function. The time is a long integer that specifies the elapsed time, in milliseconds, from the time the system was started to the time the message was created (that is, placed in the thread's message queue).

```
GetMessageTime: procedure;
    @stdcall;
    @returns( "eax" );
    @external( "__imp__GetMessageTime@0" );
```

Parameters

This function has no parameters.

Return Values

The return value specifies the message time.

Remarks

The return value from the GetMessageTime function does not necessarily increase between subsequent messages, because the value wraps to zero if the timer count exceeds the maximum value for a long integer.

To calculate time delays between messages, verify that the time of the second message is greater than the time of the first message; then, subtract the time of the first message from the time of the second message.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.216 GetMonitorInfo

The GetMonitorInfo function retrieves information about a display monitor.

```
GetMonitorInfo: procedure
(
        hMonitor    :dword;
    var lpmi        :MONITORINFO
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__GetMonitorInfoA@8" );
```

Parameters

hMonitor

[in] Handle to the display monitor of interest.

lpmi

[out] Pointer to a MONITORINFO or MONITORINFOEX structure that receives information about the specified display monitor.

You must set the cbSize member of the structure to sizeof(MONITORINFO) or sizeof(MONITORINFOEX) before calling the GetMonitorInfo function. Doing so lets the function determine the type of structure you are passing to it.

The MONITORINFOEX structure is a superset of the MONITORINFO structure. It has one additional member: a string that contains a name for the display monitor. Most applications have no use for a display monitor name, and so can save some bytes by using a MONITORINFO structure.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Windows NT/2000: To get extended error information, call GetLastError.

Requirements

Windows NT/2000: Requires Windows 2000 or later.

Windows 95/98: Requires Windows 98 or later.

3.217 GetNextDlgGroupItem

The GetNextDlgGroupItem function retrieves a handle to the first control in a group of controls that precedes (or follows) the specified control in a dialog box.

```
GetNextDlgGroupItem: procedure
(
    hDlg            :dword;
    hCtl            :dword;
    bPrevious       :boolean
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__GetNextDlgGroupItem@12" );
```

Parameters

hDlg

[in] Handle to the dialog box being searched.

hCtl

[in] Handle to the control to be used as the starting point for the search. If this parameter is NULL, the function uses the last (or first) control in the dialog box as the starting point for the search.

bPrevious

[in] Specifies how the function is to search the group of controls in the dialog box. If this parameter is TRUE, the function searches for the previous control in the group. If it is FALSE, the function searches for the next control in the group.

Return Values

If GetNextDlgGroupItem succeeds, the return value is a handle to the previous (or next) control in the group of controls.

If the function fails, the return value is NULL. To get extended error information, call GetLastError.

Remarks

The GetNextDlgGroupItem function searches controls in the order (or reverse order) they were created in the dialog box template. The first control in the group must have the WS_GROUP style; all other controls in the group must have been consecutively created and must not have the WS_GROUP style.

When searching for the previous control, the function returns the first control it locates that is visible and not disabled. If the control specified by hCtl has the WS_GROUP style, the function temporarily reverses the search to locate the first control having the WS_GROUP style, then resumes the search in the original direction, returning the first control it locates that is visible and not disabled, or returning hwndCtrl if no such control is found.

When searching for the next control, the function returns the first control it locates that is visible, not disabled, and does not have the WS_GROUP style. If it encounters a control having the WS_GROUP style, the function reverses the search, locates the first control having the WS_GROUP style, and returns this control if it is visible and not disabled. Otherwise, the function resumes the search in the original direction and returns the first control it locates that is visible and not disabled, or returns hCtl if no such control is found.

If the search for the next control in the group encounters a window with the WS_EX_CONTROLPARENT style, the system recursively searches the window's children.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.218 GetNextDlgTabItem

The GetNextDlgTabItem function retrieves a handle to the first control that has the WS_TABSTOP style that precedes (or follows) the specified control.

```
GetNextDlgTabItem: procedure
(
    hDlg            :dword;
    hCtl            :dword;
    bPrevious       :boolean
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__GetNextDlgTabItem@12" );
```

Parameters

hDlg

[in] Handle to the dialog box to be searched.

hCtl

[in] Handle to the control to be used as the starting point for the search. If this parameter is NULL, the function uses the last (or first) control in the dialog box as the starting point for the search.

bPrevious

[in] Specifies how the function is to search the dialog box. If this parameter is TRUE, the function searches for the previous control in the dialog box. If this parameter is FALSE, the function searches for the next control in the dialog box.

Return Values

If the function succeeds, the return value is the window handle of the previous (or next) control that has the WS_TABSTOP style set.

If the function fails, the return value is NULL. To get extended error information, call GetLastError.

Remarks

The GetNextDlgTabItem function searches controls in the order (or reverse order) they were created in the dialog box template. The function returns the first control it locates that is visible, not disabled, and has the WS_TABSTOP style. If no such control exists, the function returns hCtl.

If the search for the next control with the WS_TABSTOP style encounters a window with the WS_EX_CONTROLPARENT style, the system recursively searches the window's children.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.219 GetOpenClipboardWindow

The GetOpenClipboardWindow function retrieves the handle to the window that currently has the clipboard open.

```
GetOpenClipboardWindow: procedure;
    @stdcall;
    @returns( "eax" );
    @external( "__imp__GetOpenClipboardWindow@0" );
```

Parameters

This function has no parameters.

Return Values

If the function succeeds, the return value is the handle to the window that has the clipboard open. If no window has the clipboard open, the return value is NULL. To get extended error information, call GetLastError.

Remarks

If an application or dynamic-link library (DLL) specifies a NULL window handle when calling the OpenClipboard function, the clipboard is opened but is not associated with a window. In such a case, GetOpenClipboardWindow returns NULL.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.220 GetParent

The GetParent function retrieves a handle to the specified window's parent or owner.

To retrieve a handle to a specified ancestor, use the GetAncestor function.

```
GetParent: procedure
(
    hWnd            :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__GetParent@4" );
```

Parameters

hWnd

[in] Handle to the window whose parent window handle is to be retrieved.

Return Values

If the window is a child window, the return value is a handle to the parent window. If the window is a top-level window, the return value is a handle to the owner window. If the window is a top-level unowned window or if the function fails, the return value is NULL. To get extended error information, call GetLastError.

Remarks

Note that, despite its name, this function can return an owner window instead of a parent window. To obtain the

parent window and not the owner, use GetAncestor with the GA_PARENT flag.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.221 GetPriorityClipboardFormat

The GetPriorityClipboardFormat function retrieves the first available clipboard format in the specified list.

```
GetPriorityClipboardFormat: procedure
(
    var paFormatPriorityList    :var;
        cFormats                :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__GetPriorityClipboardFormat@8" );
```

Parameters

paFormatPriorityList

[in] Pointer to an array of unsigned integers identifying clipboard formats, in priority order. For a description of the standard clipboard formats, see Standard Clipboard Formats.

cFormats

[in] Specifies the number of entries in the paFormatPriorityList array. This value must not be greater than the number of entries in the list.

Return Values

If the function succeeds, the return value is the first clipboard format in the list for which data is available. If the clipboard is empty, the return value is NULL. If the clipboard contains data, but not in any of the specified formats, the return value is –1. To get extended error information, call GetLastError.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.222 GetProcessWindowStation

The GetProcessWindowStation function retrieves a handle to the window station associated with the calling process.

```
GetProcessWindowStation: procedure;
    @stdcall;
    @returns( "eax" );
    @external( "__imp__GetProcessWindowStation@0" );
```

Parameters

This function has no parameters.

Return Values

If the function succeeds, the return value is a handle to the window station associated with the calling process.

If the function fails, the return value is NULL. This can occur if the calling process is not an application written for Windows NT/Windows 2000. To get extended error information, call GetLastError.

Remarks

The system associates a window station with a process when the process is created. A process can use the SetProcessWindowStation function to change its window station.

The calling process can use the returned handle in calls to the GetUserObjectInformation, GetUserObjectSecurity, SetUserObjectInformation, and SetUserObjectSecurity functions.

Windows NT version 3.51: A service application does not have an associated window station or desktop until the service calls a USER or GDI function that interacts with the desktop. If a service calls GetProcessWindowStation before it has an associated window station, the return value is NULL. After a service interacts with the desktop, the return value is a valid window station handle.

Windows NT version 4.0: A service application is created with an associated window station and desktop, so there is no need to call a USER or GDI function to connect the service to a window station and desktop.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Unsupported.

3.223 GetProp

The GetProp function retrieves a data handle from the property list of the specified window. The character string identifies the handle to be retrieved. The string and handle must have been added to the property list by a previous call to the SetProp function.

```
GetProp: procedure
(
    hWnd            :dword;
    lpString        :string
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__GetPropA@8" );
```

Parameters

hWnd

[in] Handle to the window whose property list is to be searched.

lpString

[in] Pointer to a null-terminated character string or contains an atom that identifies a string. If this parameter is an atom, it must have been created by using the GlobalAddAtom function. The atom, a 16-bit value, must be placed in the low-order word of the lpString parameter; the high-order word must be zero.

Return Values

If the property list contains the string, the return value is the associated data handle. Otherwise, the return value is NULL.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.224 GetQueueStatus

The GetQueueStatus function indicates the type of messages found in the calling thread's message queue.

```
GetQueueStatus: procedure
(
    flags           :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__GetQueueStatus@4" );
```

Parameters

flags

[in] Specifies the types of messages for which to check. This parameter can be one or more of the following values.

Value	Meaning
QS_ALLEVENTS	An input, WM_TIMER, WM_PAINT, WM_HOTKEY, or posted message is in the queue.
QS_ALLINPUT	Any message is in the queue.
QS_ALLPOSTMESSAGE	A posted message (other than those listed here) is in the queue.
QS_HOTKEY	A WM_HOTKEY message is in the queue.
QS_INPUT	An input message is in the queue.
QS_KEY	A WM_KEYUP, WM_KEYDOWN, WM_SYSKEYUP, or WM_SYSKEYDOWN message is in the queue.
QS_MOUSE	A WM_MOUSEMOVE message or mouse-button message (WM_LBUTTONUP, WM_RBUTTONDOWN, and so on).
QS_MOUSEBUTTON	A mouse-button message (WM_LBUTTONUP, WM_RBUTTONDOWN, and so on).
QS_MOUSEMOVE	A WM_MOUSEMOVE message is in the queue.
QS_PAINT	A WM_PAINT message is in the queue.
QS_POSTMESSAGE	A posted message (other than those listed here) is in the queue.
QS_SENDMESSAGE	A message sent by another thread or application is in the queue.
QS_TIMER	A WM_TIMER message is in the queue.

Return Values

The high-order word of the return value indicates the types of messages currently in the queue. The low-order word indicates the types of messages that have been added to the queue and that are still in the queue since the last call to the GetQueueStatus, GetMessage, or PeekMessage function.

Remarks

The presence of a QS_ flag in the return value does not guarantee that a subsequent call to the GetMessage or PeekMessage function will return a message. GetMessage and PeekMessage perform some internal filtering that may cause the message to be processed internally. For this reason, the return value from GetQueueStatus should be considered only a hint as to whether GetMessage or PeekMessage should be called.

The QS_ALLPOSTMESSAGE and QS_POSTMESSAGE flags differ in when they are cleared. QS_POSTMESSAGE is cleared when you call GetMessage or PeekMessage, whether or not you are filtering messages. QS_ALLPOSTMESSAGE is cleared when you call GetMessage or PeekMessage without filtering messages (wMsgFilterMin and wMsgFilterMax are 0). This can be useful when you call PeekMessage multiple times to get messages in different ranges.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.225 GetScrollBarInfo

The GetScrollBarInfo function retrieves information about the specified scroll bar.

```
GetScrollBarInfo: procedure
(
        hwnd        :dword;
        idObject    :dword;
    var psbi        :SCROLLBARINFO
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__GetScrollBarInfo@12" );
```

Parameters

hwnd

[in] Handle to a window associated with the scroll bar whose information is to be retrieved. If the idObject parameter is OBJID_CLIENT, hwnd is a handle to a scroll bar control. Otherwise, hwnd is a handle to a window created with WS_VSCROLL and/or WS_HSCROLL style.

idObject

[in] Specifies the scroll bar object. This parameter can be one of the following values.

Value	Meaning
OBJID_CLIENT	The hwnd parameter is a handle to a scroll bar control.
OBJID_HSCROLL	The horizontal scroll bar of the hwnd window.
OBJID_VSCROLL	The vertical scroll bar of the hwnd window.

psbi

[out] Pointer to a SCROLLBARINFO structure to receive the information. Before calling GetScrollBarInfo, set the cbSize member to sizeof(SCROLLBARINFO).

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Requirements

Windows NT/2000: Requires Windows NT 4.0 SP6 or later.

Windows 95/98: Requires Windows 98 or later.

3.226 GetScrollInfo

The GetScrollInfo function retrieves the parameters of a scroll bar, including the minimum and maximum scrolling positions, the page size, and the position of the scroll box (thumb).

```
GetScrollInfo: procedure
(
        hwnd        :dword;
        fnBar       :dword;
    var lpsi        :SCROLLINFO
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__GetScrollInfo@12" );
```

Parameters

hwnd

[in] Handle to a scroll bar control or a window with a standard scroll bar, depending on the value of the fnBar parameter.

fnBar

[in] Specifies the type of scroll bar for which to retrieve parameters. This parameter can be one of the following values.

Value	Meaning
SB_CTL	Retrieves the parameters for a scroll bar control. The hwnd parameter must be the handle to the scroll bar control.
SB_HORZ	Retrieves the parameters for the window's standard horizontal scroll bar.
SB_VERT	Retrieves the parameters for the window's standard vertical scroll bar.

lpsi

[in/out] Pointer to a SCROLLINFO structure. Before calling GetScrollInfo, set the cbSize member of the structure to sizeof(SCROLLINFO), and set the fMask member to specify the scroll bar parameters to retrieve. Before returning, the function copies the specified parameters to the appropriate members of the structure.

The fMask member can be one or more of the following values.

Value	Meaning
SIF_PAGE	Copies the scroll page to the nPage member of the SCROLLINFO structure pointed to by lpsi.
SIF_POS	Copies the scroll position to the nPos member of the SCROLLINFO structure pointed to by lpsi.
SIF_RANGE	Copies the scroll range to the nMin and nMax members of the SCROLLINFO structure pointed to by lpsi.
SIF_TRACKPOS	Copies the current scroll box tracking position to the nTrackPos member of the SCROLLINFO structure pointed to by lpsi.

Return Values

If the function retrieved any values, the return value is nonzero.

If the function does not retrieve any values, the return value is zero. To get extended error information, call GetLastError.

Remarks

The GetScrollInfo function enables applications to use 32-bit scroll positions. Although the messages that indicate scroll-bar position, WM_HSCROLL and WM_VSCROLL, provide only 16 bits of position data, the functions SetScrollInfo and GetScrollInfo provide 32 bits of scroll-bar position data. Thus, an application can call GetScrollInfo while processing either the WM_HSCROLL or WM_VSCROLL messages to obtain 32-bit scroll-bar position data.

To get the 32-bit position of the scroll box (thumb) during a SB_THUMBTRACK request code in a WM_HSCROLL or WM_VSCROLL message, call GetScrollInfo with the SIF_TRACKPOS value in the fMask member of the SCROLLINFO structure. The function returns the tracking position of the scroll box in the nTrackPos member of the SCROLLINFO structure. This allows you to get the position of the scroll box as the user moves it. The following sample code illustrates the technique.

```
SCROLLINFO si;
case WM_HSCROLL:
    switch(LOWORD(wparam)) {
        case SB_THUMBTRACK:
          // Initialize SCROLLINFO structure

            ZeroMemory(&si, sizeof(SCROLLINFO));
            si.cbSize = sizeof(SCROLLINFO);
            si.fMask = SIF_TRACKPOS;

          // Call GetScrollInfo to get current tracking
          //    position in si.nTrackPos

            if (!GetScrollInfo(hwnd, SB_HORZ, &si) )
                return 1; // GetScrollInfo failed
            break;
        .
        .
        .
    }
```

Requirements

Windows NT/2000: Requires Windows NT 3.51 or later.

Windows 95/98: Requires Windows 95 or later.

3.227 GetScrollPos

The GetScrollPos function retrieves the current position of the scroll box (thumb) in the specified scroll bar. The current position is a relative value that depends on the current scrolling range. For example, if the scrolling range is 0 through 100 and the scroll box is in the middle of the bar, the current position is 50.

Note The GetScrollPos function is provided for backward compatibility. New applications should use the GetScrollInfo function.

```
GetScrollPos: procedure
```

```
(
    hWnd            :dword;
    nBar            :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__GetScrollPos@8" );
```

Parameters

hWnd

[in] Handle to a scroll bar control or a window with a standard scroll bar, depending on the value of the nBar parameter.

nBar

[in] Specifies the scroll bar to be examined. This parameter can be one of the following values.

Value	Meaning
SB_CTL	Retrieves the position of the scroll box in a scroll bar control. The hWnd parameter must be the handle to the scroll bar control.
SB_HORZ	Retrieves the position of the scroll box in a window's standard horizontal scroll bar.
SB_VERT	Retrieves the position of the scroll box in a window's standard vertical scroll bar.

Return Values

If the function succeeds, the return value is the current position of the scroll box.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

The GetScrollPos function enables applications to use 32-bit scroll positions. Although the messages that indicate scroll bar position, WM_HSCROLL and WM_VSCROLL, are limited to 16 bits of position data, the functions SetScrollPos, SetScrollRange, GetScrollPos, and GetScrollRange support 32-bit scroll bar position data. Thus, an application can call GetScrollPos while processing either the WM_HSCROLL or WM_VSCROLL messages to obtain 32-bit scroll bar position data.

To get the 32-bit position of the scroll box (thumb) during a SB_THUMBTRACK request code in a WM_HSCROLL or WM_VSCROLL message, use the GetScrollInfo function.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.228 GetScrollRange

The GetScrollRange function retrieves the current minimum and maximum scroll box (thumb) positions for the specified scroll bar.

Note The GetScrollRange function is provided for compatibility only. New applications should use the GetScrollInfo function.

```
GetScrollRange: procedure
(
```

```
        hWnd        :dword;
        nBar        :dword;
    var lpMinPos    :dword;
    var lpMaxPos    :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__GetScrollRange@16" );
```

Parameters

hWnd

[in] Handle to a scroll bar control or a window with a standard scroll bar, depending on the value of the nBar parameter.

nBar

[in] Specifies the scroll bar from which the positions are retrieved. This parameter can be one of the following values.

Value	Meaning
SB_CTL	Retrieves the positions of a scroll bar control. The hWnd parameter must be the handle to the scroll bar control.
SB_HORZ	Retrieves the positions of the window's standard horizontal scroll bar.
SB_VERT	Retrieves the positions of the window's standard vertical scroll bar.

lpMinPos

[out] Pointer to the integer variable that receives the minimum position.

lpMaxPos

[out] Pointer to the integer variable that receives the maximum position.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

If the specified window does not have standard scroll bars or is not a scroll bar control, the GetScrollRange function copies zero to the lpMinPos and lpMaxPos parameters.

The default range for a standard scroll bar is 0 through 100. The default range for a scroll bar control is empty (both values are zero).

The messages that indicate scroll bar position, WM_HSCROLL and WM_VSCROLL, are limited to 16 bits of position data. However, because SetScrollInfo, SetScrollPos, SetScrollRange, GetScrollInfo, GetScrollPos, and GetScrollRange support 32-bit scroll bar position data, there is a way to circumvent the 16-bit barrier of the WM_HSCROLL and WM_VSCROLL messages. See the GetScrollInfo function for a description of the technique.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.229 GetSubMenu

The GetSubMenu function retrieves a handle to the drop-down menu or submenu activated by the specified menu item.

```
GetSubMenu: procedure
(
    hMenu           :dword;
    nPos            :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__GetSubMenu@8" );
```

Parameters

hMenu

[in] Handle to the menu.

nPos

[in] Specifies the zero-based relative position in the specified menu of an item that activates a drop-down menu or submenu.

Return Values

If the function succeeds, the return value is a handle to the drop-down menu or submenu activated by the menu item. If the menu item does not activate a drop-down menu or submenu, the return value is NULL.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.230 GetSysColor

The GetSysColor function retrieves the current color of the specified display element. Display elements are the parts of a window and the display that appear on the system display screen.

```
GetSysColor: procedure
(
    nIndex          :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__GetSysColor@4" );
```

Parameters

nIndex

[in] Specifies the display element whose color is to be retrieved. This parameter can be one of the following values.

Value	Meaning
COLOR_3DDKSHADOW	Dark shadow for three-dimensional display elements.

COLOR_3DFACE, COLOR_BTNFACE	Face color for three-dimensional display elements and for dialog box backgrounds.
COLOR_3DHILIGHT, COLOR_3DHIGHLIGHT, COLOR_BTNHILIGHT, COLOR_BTNHIGHLIGHT	Highlight color for three-dimensional display elements (for edges facing the light source.)
COLOR_3DLIGHT	Light color for three-dimensional display elements (for edges facing the light source.)
COLOR_3DSHADOW, COLOR_BTNSHADOW	Shadow color for three-dimensional display elements (for edges facing away from the light source).
COLOR_ACTIVEBORDER	Active window border.
COLOR_ACTIVECAPTION	Active window title bar. Windows 98, Windows 2000: Specifies the left side color in the color gradient of an active window's title bar if the gradient effect is enabled.
COLOR_APPWORKSPACE	Background color of multiple document interface (MDI) applications.
COLOR_BACKGROUND, COLOR_DESKTOP	Desktop.
COLOR_BTNTEXT	Text on push buttons.
COLOR_CAPTIONTEXT	Text in caption, size box, and scroll bar arrow box.
COLOR_GRADIENTACTIVECAPTION	Windows 98, Windows 2000: Right side color in the color gradient of an active window's title bar. COLOR_ACTIVECAPTION specifies the left side color. Use SPI_GETGRADIENTCAPTIONS with the System-ParametersInfo function to determine whether the gradient effect is enabled.
COLOR_GRADIENTINACTIVECAPTION	Windows 98, Windows 2000: Right side color in the color gradient of an inactive window's title bar. COLOR_INACTIVECAPTION specifies the left side color.
COLOR_GRAYTEXT	Grayed (disabled) text. This color is set to 0 if the current display driver does not support a solid gray color.
COLOR_HIGHLIGHT	Item(s) selected in a control.
COLOR_HIGHLIGHTTEXT	Text of item(s) selected in a control.
COLOR_HOTLIGHT	Windows 98, Windows 2000: Color for a hot-tracked item. Single clicking a hot-tracked item executes the item.
COLOR_INACTIVEBORDER	Inactive window border.
COLOR_INACTIVECAPTION	Inactive window caption. Windows 98, Windows 2000: Specifies the left side color in the color gradient of an inactive window's title bar if the gradient effect is enabled.
COLOR_INACTIVECAPTIONTEXT	Color of text in an inactive caption.
COLOR_INFOBK	Background color for tooltip controls.
COLOR_INFOTEXT	Text color for tooltip controls.
COLOR_MENU	Menu background.
COLOR_MENUHILIGHT	Whistler: The color used to highlight menu items when the menu appears as a flat menu (see SystemParametersInfo). The highlighted menu item is outlined with COLOR_HIGHLIGHT.
COLOR_MENUBAR	Whistler: The background color for the menu bar when menus appear as flat menus (see SystemParametersInfo). However, COLOR_MENU continues to specify the background color of the menu popup.
COLOR_MENUTEXT	Text in menus.
COLOR_SCROLLBAR	Scroll bar gray area.

COLOR_WINDOW	Window background.
COLOR_WINDOWFRAME	Window frame.
COLOR_WINDOWTEXT	Text in windows.

Return Values

The function returns the red, green, blue (RGB) color value of the given element.

If the nIndex parameter is out of range, the return value is zero. Because zero is also a valid RGB value, you cannot use GetSysColor to determine whether a system color is supported by the current platform. Instead, use the GetSysColorBrush function, which returns NULL if the color is not supported.

Remarks

System colors for monochrome displays are usually interpreted as shades of gray.

To paint with a system color brush, an application should use GetSysColorBrush(nIndex) instead of CreateSolidBrush(GetSysColor(nIndex)), because GetSysColorBrush returns a cached brush (instead of allocating a new one).

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.231 GetSysColorBrush

The GetSysColorBrush function retrieves a handle identifying a logical brush that corresponds to the specified color index.

```
GetSysColorBrush: procedure
(
    nIndex          :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__GetSysColorBrush@4" );
```

Parameters

nIndex

[in] Specifies a color index. This value corresponds to the color used to paint one of the window elements.

Return Values

The return value identifies a logical brush if the nIndex parameter is supported by the current platform. Otherwise, it returns NULL.

Remarks

A brush is a bitmap that the system uses to paint the interiors of filled shapes. An application can retrieve the current system colors by calling the GetSysColor function. An application can set the current system colors by calling the SetSysColors function.

An application must not register a window class for a window using a system brush. To register a window class with a system color, see the documentation of the hbrBackground member of the WNDCLASS or WNDCLASSEX structures.

System color brushes track changes in system colors. In other words, when the user changes a system color, the associated system color brush automatically changes to the new color.

To paint with a system color brush, an application should use GetSysColorBrush(nIndex) instead of CreateSolid-Brush(GetSysColor(nIndex)), because GetSysColorBrush returns a cached brush instead of allocating a new one.

System color brushes are owned by the system and must not be destroyed.

Requirements

Windows NT/2000: Requires Windows NT 3.51 or later.

Windows 95/98: Requires Windows 95 or later.

3.232 GetSystemMenu

The GetSystemMenu function allows the application to access the window menu (also known as the system menu or the control menu) for copying and modifying.

```
GetSystemMenu: procedure
(
    hWnd            :dword;
    bRevert         :boolean
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__GetSystemMenu@8" );
```

Parameters

hWnd

[in] Handle to the window that will own a copy of the window menu.

bRevert

[in] Specifies the action to be taken. If this parameter is FALSE, GetSystemMenu returns a handle to the copy of the window menu currently in use. The copy is initially identical to the window menu, but it can be modified.

If this parameter is TRUE, GetSystemMenu resets the window menu back to the default state. The previous window menu, if any, is destroyed.

Return Values

If the bRevert parameter is FALSE, the return value is a handle to a copy of the window menu. If the bRevert parameter is TRUE, the return value is NULL.

Remarks

Any window that does not use the GetSystemMenu function to make its own copy of the window menu receives the standard window menu.

The window menu initially contains items with various identifier values, such as SC_CLOSE, SC_MOVE, and SC_SIZE.

Menu items on the window menu send WM_SYSCOMMAND messages.

All predefined window menu items have identifier numbers greater than 0xF000. If an application adds commands to the window menu, it should use identifier numbers less than 0xF000.

The system automatically grays items on the standard window menu, depending on the situation. The application can perform its own checking or graying by responding to the WM_INITMENU message that is sent before any menu is displayed.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.233 GetSystemMetrics

The GetSystemMetrics function retrieves various system metrics (widths and heights of display elements) and system configuration settings. All dimensions retrieved by GetSystemMetrics are in pixels.

```
GetSystemMetrics: procedure
(
    nIndex          :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__GetSystemMetrics@4" );
```

Parameters

nIndex

[in] Specifies the system metric or configuration setting to retrieve. All SM_CX* values are widths. All SM_CY* values are heights. The following values are defined.

Value	Meaning
SM_ARRANGE	Flags specifying how the system arranged minimized windows. For more information about minimized windows, see the following Remarks section.
SM_CLEANBOOT	Value that specifies how the system was started: 0 Normal boot 1 Fail-safe boot 2 Fail-safe with network boot Fail-safe boot (also called SafeBoot) bypasses the user's startup files.
SM_CMONITORS	Windows 98, Windows 2000: Number of display monitors on the desktop. See Remarks for more information.
SM_CMOUSEBUTTONS	Number of buttons on mouse, or zero if no mouse is installed.
SM_CXBORDER, SM_CYBORDER	Width and height, in pixels, of a window border. This is equivalent to the SM_CXEDGE value for windows with the 3-D look.
SM_CXCURSOR, SM_CYCURSOR	Width and height, in pixels, of a cursor. The system cannot create cursors of other sizes.
SM_CXDLGFRAME, SM_CYDLGFRAME	Same as SM_CXFIXEDFRAME and SM_CYFIXEDFRAME.

SM_CXDOUBLECLK, SM_CYDOUBLECLK	Width and height, in pixels, of the rectangle around the location of a first click in a double-click sequence. The second click must occur within this rectangle for the system to consider the two clicks a double-click. (The two clicks must also occur within a specified time.) To set the width and height of the double-click rectangle, call SystemParametersInfo with the SPI_SETDOUBLECLKHEIGHT and SPI_SETDOUBLECLKWIDTH flags.
SM_CXDRAG, SM_CYDRAG	Width and height, in pixels, of a rectangle centered on a drag point to allow for limited movement of the mouse pointer before a drag operation begins. This allows the user to click and release the mouse button easily without unintentionally starting a drag operation.
SM_CXEDGE, SM_CYEDGE	Dimensions, in pixels, of a 3-D border. These are the 3-D counterparts of SM_CXBORDER and SM_CYBORDER.
SM_CXFIXEDFRAME, SM_CYFIXEDFRAME	Thickness, in pixels, of the frame around the perimeter of a window that has a caption but is not sizable. SM_CXFIXEDFRAME is the heightof the horizontal border and SM_CYFIXEDFRAME is the width of the vertical border. Same as SM_CXDLGFRAME and SM_CYDLGFRAME.
SM_CXFOCUSBORDER, SM_CYFOCUSBORDER	Whistler: The width of the left and right edges and the height of the top and bottom edges of the focus rectangle drawn by DrawFocusRect. These values are in pixels.
SM_CXFRAME, SM_CYFRAME	Same as SM_CXSIZEFRAME and SM_CYSIZEFRAME.
SM_CXFULLSCREEN, SM_CYFULLSCREEN	Width and height of the client area for a full-screen window on the primary display monitor. To get the coordinates of the portion of the screen not obscured by the system taskbar or by application desktop toolbars, call the SystemParameters-Info function with the SPI_GETWORKAREA value.
SM_CXHSCROLL, SM_CYHSCROLL	Width, in pixels, of the arrow bitmap on a horizontal scroll bar; and height, in pixels, of a horizontal scroll bar.
SM_CXHTHUMB	Width, in pixels, of the thumb box in a horizontal scroll bar.
SM_CXICON, SM_CYICON	The default width and height, in pixels, of an icon. The LoadIcon function can load only icons of these dimensions.
SM_CXICONSPACING, SM_CYICONSPACING	Dimensions, in pixels, of a grid cell for items in large icon view. Each item fits into a rectangle of this size when arranged. These values are always greater than or equal to SM_CXICON and SM_CYICON.
SM_CXMAXIMIZED, SM_CYMAXIMIZED	Default dimensions, in pixels, of a maximized top-level window on the primary display monitor.

SM_CXMAXTRACK, SM_CYMAXTRACK	Default maximum dimensions, in pixels, of a window that has a caption and sizing borders. This metric refers to the entire desktop. The user cannot drag the window frame to a size larger than these dimensions. A window can override these values by processing the WM_GETMINMAXINFO message.
SM_CXMENUCHECK, SM_CYMENUCHECK	Dimensions, in pixels, of the default menu check-mark bitmap.
SM_CXMENUSIZE, SM_CYMENUSIZE	Dimensions, in pixels, of menu bar buttons, such as the child window close button used in the multiple document interface.
SM_CXMIN, SM_CYMIN	Minimum width and height, in pixels, of a window.
SM_CXMINIMIZED, SM_CYMINIMIZED	Dimensions, in pixels, of a normal minimized window.
SM_CXMINSPACING SM_CYMINSPACING	Dimensions, in pixels, of a grid cell for minimized windows. Each minimized window fits into a rectangle this size when arranged. These values are always greater than or equal to SM_CXMINIMIZED and SM_CYMINIMIZED.
SM_CXMINTRACK, SM_CYMINTRACK	Minimum tracking width and height, in pixels, of a window. The user cannot drag the window frame to a size smaller than these dimensions. A window can override these values by processing the WM_GETMINMAXINFO message.
SM_CXSCREEN, SM_CYSCREEN	Width and height, in pixels, of the screen of the primary display monitor. These are the same values you obtain by calling GetDeviceCaps(hdcPrimaryMonitor, HORZRES/VERTRES).
SM_CXSIZE, SM_CYSIZE	Width and height, in pixels, of a button in a window's caption or title bar.
SM_CXSIZEFRAME, SM_CYSIZEFRAME	Thickness, in pixels, of the sizing border around the perimeter of a window that can be resized. SM_CXSIZEFRAME is the width of the horizontal border, and SM_CYSIZEFRAME is the height of the vertical border.
	Same as SM_CXFRAME and SM_CYFRAME.
SM_CXSMICON, SM_CYSMICON	Recommended dimensions, in pixels, of a small icon. Small icons typically appear in window captions and in small icon view.
SM_CXSMSIZE SM_CYSMSIZE	Dimensions, in pixels, of small caption buttons.
SM_CXVIRTUALSCREEN, SM_CYVIRTUALSCREEN	Windows 98, Windows 2000: Width and height, in pixels, of the virtual screen. The virtual screen is the bounding rectangle of all display monitors. The SM_XVIRTUALSCREEN, SM_YVIRTUALSCREEN metrics are the coordinates of the top-left corner of the virtual screen.

SM_CXVSCROLL,	Width, in pixels, of a vertical scroll bar; and height, in pixels, of the arrow bitmap on a vertical scroll bar.
SM_CYVSCROLL	
SM_CYCAPTION	Height, in pixels, of a normal caption area.
SM_CYKANJIWINDOW	For double byte character set versions of the system, this is the height, in pixels, of the Kanji window at the bottom of the screen.
SM_CYMENU	Height, in pixels, of a single-line menu bar.
SM_CYSMCAPTION	Height, in pixels, of a small caption.
SM_CYVTHUMB	Height, in pixels, of the thumb box in a vertical scroll bar.
SM_DBCSENABLED	TRUE or nonzero if the double-byte character-set (DBCS) version of User.exe is installed; FALSE or zero otherwise.
SM_DEBUG	TRUE or nonzero if the debugging version of User.exe is installed; FALSE or zero otherwise.
SM_IMMENABLED	Windows 2000: TRUE or nonzero if Input Method Manager/Input Method Editor features are enabled; FALSE or zero otherwise. SM_IMMENABLED indicates whether the system is ready to use a Unicode-based IME on a Unicode application. To ensure that a language-dependent IME works, check SM_DBCSENABLED and the system ANSI code page. Otherwise the ANSI-to-Unicode conversion may not be performed correctly, or some components like fonts or registry setting may not be present.
SM_MENUDROPALIGNMENT	TRUE or nonzero if drop-down menus are right-aligned with the corresponding menu-bar item; FALSE or zero if the menus are left-aligned.
SM_MIDEASTENABLED	TRUE if the system is enabled for Hebrew and Arabic languages.
SM_MOUSEPRESENT	TRUE or nonzero if a mouse is installed; FALSE or zero otherwise.
SM_MOUSEWHEELPRESENT	Windows NT 4.0 and later, Windows 98: TRUE or nonzero if a mouse with a wheel is installed; FALSE or zero otherwise.
SM_NETWORK	The least significant bit is set if a network is present; otherwise, it is cleared. The other bits are reserved for future use.
SM_PENWINDOWS	TRUE or nonzero if the Microsoft Windows for Pen computing extensions are installed; FALSE or zero otherwise.
SM_REMOTESESSION	Windows NT 4.0 SP4 or later: This system metric is used in a Terminal Services environment. If the calling process is associated with a Terminal Services client session, the return value is TRUE or nonzero. If the calling process is associated with the Terminal Server console session, the return value is zero.
SM_SECURE	TRUE if security is present; FALSE otherwise.
SM_SAMEDISPLAYFORMAT	Windows 98, Windows 2000: TRUE if all the display monitors have the same color format, FALSE otherwise. Note that two displays can have the same bit depth, but different color formats. For example, the red, green, and blue pixels can be encoded with different numbers of bits, or those bits can be located in different places in a pixel's color value.
SM_SHOWSOUNDS	TRUE or nonzero if the user requires an application to present information visually in situations where it would otherwise present the information only in audible form; FALSE, or zero, otherwise.

SM_SLOWMACHINE	TRUE if the computer has a low-end (slow) processor; FALSE otherwise.
SM_SWAPBUTTON	TRUE or nonzero if the meanings of the left and right mouse buttons are swapped; FALSE or zero otherwise.
SM_XVIRTUALSCREEN, SM_YVIRTUALSCREEN	Windows 98, Windows 2000: Coordinates for the left side and the top of the virtual screen. The virtual screen is the bounding rectangle of all display monitors. The SM_CXVIRTUALSCREEN, SM_CYVIRTUALSCREEN metrics are the width and height of the virtual screen.

Return Values

If the function succeeds, the return value is the requested system metric or configuration setting.

If the function fails, the return value is zero. GetLastError does not provide extended error information.

Remarks

System metrics may vary from display to display.

GetSystemMetrics(SM_CMONITORS) counts only display monitors. This is different from EnumDisplayMonitors which enumerates display monitors and also non-display pseudo-monitors.

The SM_ARRANGE setting specifies how the system arranges minimized windows, and consists of a starting position and a direction. The starting position can be one of the following values.

Value	Meaning
ARW_BOTTOMLEFT	Start at the lower-left corner of the screen (default position).
ARW_BOTTOMRIGHT	Start at the lower-right corner of the screen. Equivalent to ARW_STARTRIGHT.
ARW_HIDE	Hide minimized windows by moving them off the visible area of the screen.
ARW_TOPLEFT	Start at the upper-left corner of the screen. Equivalent to ARV_STARTTOP.
ARW_TOPRIGHT	Start at the upper-right corner of the screen. Equivalent to ARW_STARTTOP | SRW_STARTRIGHT.

The direction in which to arrange can be one of the following values.

Value	Meaning
ARW_DOWN	Arrange vertically, top to bottom.
ARW_LEFT	Arrange horizontally, left to right.
ARW_RIGHT	Arrange horizontally, right to left.
ARW_UP	Arrange vertically, bottom to top.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.234 GetTabbedTextExtent

The GetTabbedTextExtent function computes the width and height of a character string. If the string contains one or more tab characters, the width of the string is based upon the specified tab stops. The GetTabbedTextExtent function uses the currently selected font to compute the dimensions of the string.

```
GetTabbedTextExtent: procedure
(
        hDC                 :dword;
```

```
        lpString            :string;
        nCount              :dword;
        nTabPositions       :dword;
    var lpnTabStopPositions :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__GetTabbedTextExtentA@20" );
```

Parameters

hDC

[in] Handle to the device context.

lpString

[in] Pointer to a character string.

nCount

[in] Specifies the length of the text string. For the ANSI function it is a BYTE count and for the Unicode function it is a WORD count. Note that for the ANSI function, characters in SBCS code pages take one byte each, while most characters in DBCS code pages take two bytes; for the Unicode function, most currently defined Unicode characters (those in the Basic Multilingual Plane (BMP)) are one WORD while Unicode surrogates are two WORDs.

Windows 95/98: This value may not exceed 8192.

nTabPositions

[in] Specifies the number of tab-stop positions in the array pointed to by the lpnTabStopPositions parameter.

lpnTabStopPositions

[in] Pointer to an array containing the tab-stop positions, in device units. The tab stops must be sorted in increasing order; the smallest x-value should be the first item in the array.

Return Values

If the function succeeds, the return value is the dimensions of the string in logical units. The height is in the high-order word and the width is in the low-order word.

If the function fails, the return value is 0. GetTabbedTextExtent will fail if hDC is invalid and if nTabPositions is less than 0.

Windows NT/ 2000: To get extended error information, call GetLastError.

Remarks

The current clipping region does not affect the width and height returned by the GetTabbedTextExtent function.

Because some devices do not place characters in regular cell arrays (that is, they kern the characters), the sum of the extents of the characters in a string may not be equal to the extent of the string.

If the nTabPositions parameter is zero and the lpnTabStopPositions parameter is NULL, tabs are expanded to eight times the average character width.

If nTabPositions is 1, the tab stops are separated by the distance specified by the first value in the array to which lpnTabStopPositions points.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.235 GetThreadDesktop

The GetThreadDesktop function retrieves a handle to the desktop associated with a specified thread.

```
GetThreadDesktop: procedure
(
    dwThreadID          :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__GetThreadDesktop@4" );
```

Parameters

dwThreadId

[in] Handle to the thread for which to return the desktop handle. The GetCurrentThreadId and CreateProcess functions return thread identifiers.

Return Values

If the function succeeds, the return value is a handle to the desktop associated with the specified thread. You do not need to call the CloseDesktop function to close the returned handle.

If the function fails, the return value is NULL. To get extended error information, call GetLastError.

Remarks

The system associates a desktop with a thread when that thread is created. A thread can use the SetThreadDesktop function to change its desktop. The desktop associated with a thread must be on the window station associated with the thread's process.

The calling process can use the returned handle in calls to the GetUserObjectInformation, GetUserObjectSecurity, SetUserObjectInformation, and SetUserObjectSecurity functions.

Windows 95/98: The system does not support multiple desktops, so GetThreadDesktop always returns the same value.

Windows NT version 3.51: A service application does not have an associated window station or desktop until the service calls a USER or GDI function that interacts with the desktop. If a service calls GetThreadDesktop before it has an associated desktop, the return value is NULL. After a service interacts with the desktop, the return value is a valid desktop handle.

Windows NT/2000: A service application is created with an associated window station and desktop, so there is no need to call a USER or GDI function to connect the service to a window station and desktop.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.236 GetTitleBarInfo

The GetTitleBarInfo function retrieves information about the specified title bar.

```
GetTitleBarInfo: procedure
(
        hwnd            :dword;
```

```
    var pti             :TITLEBARINFO
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__GetTitleBarInfo@8" );
```

Parameters

hwnd

[in] Handle to the title bar whose information is to be retrieved.

pti

[out] Pointer to a TITLEBARINFO structure to receive the information.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Requirements

Windows NT/2000: Requires Windows NT 4.0 SP6 or later.

Windows 95/98: Requires Windows 98 or later.

3.237 GetTopWindow

The GetTopWindow function examines the Z order of the child windows associated with the specified parent window and retrieves a handle to the child window at the top of the Z order.

```
GetTopWindow: procedure
(
    hWnd            :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__GetTopWindow@4" );
```

Parameters

hWnd

[in] Handle to the parent window whose child windows are to be examined. If this parameter is NULL, the function returns a handle to the window at the top of the Z order.

Return Values

If the function succeeds, the return value is a handle to the child window at the top of the Z order. If the specified window has no child windows, the return value is NULL. To get extended error information, use the GetLastError function.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.238 GetUpdateRect

The GetUpdateRect function retrieves the coordinates of the smallest rectangle that completely encloses the update region of the specified window. GetUpdateRect retrieves the rectangle in logical coordinates. If there is no update region, GetUpdateRect retrieves an empty rectangle (sets all coordinates to zero).

```
GetUpdateRect: procedure
(
        hWnd        :dword;
    var lpRect      :RECT;
        bErase      :boolean
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__GetUpdateRect@12" );
```

Parameters

hWnd

[in] Handle to the window with an update region that is to be retrieved.

lpRect

[out] Pointer to the RECT structure that receives the coordinates of the enclosing rectangle.

An application can set this parameter to NULL to determine whether an update region exists for the window. If this parameter is NULL, GetUpdateRect returns nonzero if an update region exists, and zero if one does not. This provides a simple and efficient means of determining whether a WM_PAINT message resulted from an invalid area.

bErase

[in] Specifies whether the background in the update region is to be erased. If this parameter is TRUE and the update region is not empty, GetUpdateRect sends a WM_ERASEBKGND message to the specified window to erase the background.

Return Values

If the update region is not empty, the return value is nonzero.

If there is no update region, the return value is zero.

Windows NT/ 2000: To get extended error information, call GetLastError.

Remarks

The update rectangle retrieved by the BeginPaint function is identical to that retrieved by GetUpdateRect.

BeginPaint automatically validates the update region, so any call to GetUpdateRect made immediately after the call to BeginPaint retrieves an empty update region.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.239 GetUpdateRgn

The GetUpdateRgn function retrieves the update region of a window by copying it into the specified region. The coordinates of the update region are relative to the upper-left corner of the window (that is, they are client coordinates).

```
GetUpdateRgn: procedure
(
    hWnd            :dword;
    hRgn            :dword;
    bErase          :boolean
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__GetUpdateRgn@12" );
```

Parameters

hWnd

[in] Handle to the window with an update region that is to be retrieved.

hRgn

[in] Handle to the region to receive the update region.

bErase

[in] Specifies whether the window background should be erased and whether nonclient areas of child windows should be drawn. If this parameter is FALSE, no drawing is done.

Return Values

The return value indicates the complexity of the resulting region; it can be one of the following values.

Value	Meaning
COMPLEXREGION	Region consists of more than one rectangle.
ERROR	An error occurred.
NULLREGION	Region is empty.
SIMPLEREGION	Region is a single rectangle.

Remarks

The BeginPaint function automatically validates the update region, so any call to GetUpdateRgn made immediately after the call to BeginPaint retrieves an empty update region.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.240 GetUserObjectInformation

The GetUserObjectInformation function retrieves information about a window station or desktop object.

```
GetUserObjectInformation: procedure
(
        hObj            :dword;
        nIndex          :dword;
    var pvInfo          :var;
```

```
        nLength         :dword;
    var lpnLengthNeeded :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__GetUserObjectInformationA@20" );
```

Parameters

hObj

[in] Handle to the window station or desktop object for which to return information. This can be an HDESK or HWINSTA handle (for example, a handle returned by CreateWindowStation, OpenWindowStation, CreateDesktop, or OpenDesktop).

nIndex

[in] Specifies the object information to be retrieved. The parameter must be one of the following values:

Value	Description
UOI_FLAGS	Returns handle flags. The *pvInfo parameter must point to a USEROBJECTFLAGS structure.*
UOI_NAME	Returns a string containing the name of the object.
UOI_TYPE	Returns a string containing the type name of the object.
UOI_USER_SID	Returns the SID structure that identifies the user that is currently associated with the specified object. If no user is associated with the object, the value returned in the buffer pointed to by lpnLengthNeeded is zero. Note that SID is a variable length structure. You will usually make a call to to determine the length of the SID before retrieving its value.

pvInfo

[out] Pointer to a buffer to receive the object information.

nLength

[in] Specifies the size, in bytes, of the buffer pointed to by the pvInfo parameter.

lpnLengthNeeded

[out] Pointer to a variable receiving the number of bytes required to store the requested information. If this variable's value is greater than the value of the nLength parameter when the function returns, the function returns FALSE, and none of the information is copied to the pvInfo buffer. If the value of the variable pointed to by lpnLengthNeeded is less than or equal to the value of nLength, the entire information block is copied.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Requirements

Windows NT/2000: Requires Windows NT 3.51 or later.

Windows 95/98: Unsupported.

3.241 GetWindow

The GetWindow function retrieves a handle to a window that has the specified relationship (Z order or owner) to

the specified window.

```
GetWindow: procedure
(
    hWnd        :dword;
    uCmd        :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__GetWindow@8" );
```

Parameters

hWnd

[in] Handle to a window. The window handle retrieved is relative to this window, based on the value of the uCmd parameter.

uCmd

[in] Specifies the relationship between the specified window and the window whose handle is to be retrieved. This parameter can be one of the following values.

Value	Meaning
GW_CHILD	The retrieved handle identifies the child window at the top of the Z order, if the specified window is a parent window; otherwise, the retrieved handle is NULL. The function examines only child windows of the specified window. It does not examine descendant windows.
GW_ENABLEDPOPUP	Windows 2000: The retrieved handle identifies the enabled popup window owned by the specified window (the search uses the first such window found using GW_HWNDNEXT); otherwise, if there are no enabled popup windows, the retrieved handle is that of the specified window.
GW_HWNDFIRST	The retrieved handle identifies the window of the same type that is highest in the Z order. If the specified window is a topmost window, the handle identifies the topmost window that is highest in the Z order. If the specified window is a top-level window, the handle identifies the top-level window that is highest in the Z order. If the specified window is a child window, the handle identifies the sibling window that is highest in the Z order.
GW_HWNDLAST	The retrieved handle identifies the window of the same type that is lowest in the Z order. If the specified window is a topmost window, the handle identifies the topmost window that is lowest in the Z order. If the specified window is a top-level window, the handle identifies the top-level window that is lowest in the Z order. If the specified window is a child window, the handle identifies the sibling window that is lowest in the Z order.
GW_HWNDNEXT	The retrieved handle identifies the window below the specified window in the Z order. If the specified window is a topmost window, the handle identifies the topmost window below the specified window. If the specified window is a top-level window, the handle identifies the top-level window below the specified window. If the specified window is a child window, the handle identifies the sibling window below the specified window.

GW_HWNDPREV	The retrieved handle identifies the window above the specified window in the Z order. If the specified window is a topmost window, the handle identifies the topmost window above the specified window. If the specified window is a top-level window, the handle identifies the top-level window above the specified window. If the specified window is a child window, the handle identifies the sibling window above the specified window.
GW_OWNER	The retrieved handle identifies the specified window's owner window, if any. For more information, see Owned Windows.

Return Values

If the function succeeds, the return value is a window handle. If no window exists with the specified relationship to the specified window, the return value is NULL. To get extended error information, call GetLastError.

Remarks

The EnumChildWindows function is more reliable than calling GetWindow in a loop. An application that calls GetWindow to perform this task risks being caught in an infinite loop or referencing a handle to a window that has been destroyed.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.242 GetWindowContextHelpId

Retrieves the Help context identifier, if any, associated with the specified window.

```
GetWindowContextHelpId: procedure
(
    hwnd        :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__GetWindowContextHelpId@4" );
```

Parameters

hwnd

Handle to the window for which the Help context identifier is to be retrieved.

Return Values

Returns the Help context identifier if the window has one, or zero otherwise.

See Also

SetWindowContextHelpId

Requirements

Windows NT/2000: Requires Windows NT 3.51 or later.

Windows 95/98: Requires Windows 95 or later.

3.243 GetWindowDC

The GetWindowDC function retrieves the device context (DC) for the entire window, including title bar, menus, and scroll bars. A window device context permits painting anywhere in a window, because the origin of the device context is the upper-left corner of the window instead of the client area.

GetWindowDC assigns default attributes to the window device context each time it retrieves the device context. Previous attributes are lost.

```
GetWindowDC: procedure
(
    hWnd        :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__GetWindowDC@4" );
```

Parameters

hWnd

[in] Handle to the window with a device context that is to be retrieved. If this value is NULL, GetWindowDC retrieves the device context for the entire screen.

Windows 98, Windows 2000: If this parameter is NULL, GetWindowDC retrieves the device context for the primary display monitor. To get the device context for other display monitors, use the EnumDisplayMonitors and CreateDC functions.

Return Values

If the function succeeds, the return value is a handle to a device context for the specified window.

If the function fails, the return value is NULL, indicating an error or an invalid hWnd parameter.

Windows NT/ 2000: To get extended error information, call GetLastError.

Remarks

GetWindowDC is intended for special painting effects within a window's nonclient area. Painting in nonclient areas of any window is not recommended.

The GetSystemMetrics function can be used to retrieve the dimensions of various parts of the nonclient area, such as the title bar, menu, and scroll bars.

The GetDC function can be used to retrieve a device context for the entire screen.

After painting is complete, the ReleaseDC function must be called to release the device context. Not releasing the window device context has serious effects on painting requested by applications.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.244 GetWindowInfo

The GetWindowInfo function retrieves information about the specified window.

```
GetWindowInfo: procedure
(
```

```
        hWnd    :dword;
    var pwi     :WINDOWINFO
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__GetWindowInfo@8" );
```

Parameters

hwnd

[in] Handle to the window whose information is to be retrieved.

pwi

[out] Pointer to a WINDOWINFO structure to receive the information.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Windows NT/2000: To get extended error information, call GetLastError.

Requirements

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.

Windows 95/98: Requires Windows 98 or later.

3.245 GetWindowLong

The GetWindowLong function retrieves information about the specified window. The function also retrieves the 32-bit (long) value at the specified offset into the extra window memory.

If you are retrieving a pointer or a handle, this function has been superseded by the GetWindowLongPtr function. (Pointers and handles are 32 bits on 32-bit Windows and 64 bits on 64-bit Windows.) To write code that is compatible with both 32-bit and 64-bit versions of Windows, use GetWindowLongPtr.

```
GetWindowLong: procedure
(
    hWnd        :dword;
    nIndex      :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__GetWindowLongA@8" );
```

Parameters

hWnd

[in] Handle to the window and, indirectly, the class to which the window belongs.

nIndex

[in] Specifies the zero-based offset to the value to be retrieved. Valid values are in the range zero through the number of bytes of extra window memory, minus four; for example, if you specified 12 or more bytes of extra memory, a value of 8 would be an index to the third 32-bit integer. To retrieve any other value, specify one of the following values.

Value	Action
GWL_EXSTYLE	Retrieves the extended window styles. For more information, see CreateWindowEx.
GWL_STYLE	Retrieves the window styles.
GWL_WNDPROC	Retrieves the address of the window procedure, or a handle representing the address of the window procedure. You must use the CallWindowProc function to call the window procedure.
GWL_HINSTANCE	Retrieves a handle to the application instance.
GWL_HWNDPARENT	Retrieves a handle to the parent window, if any.
GWL_ID	Retrieves the identifier of the window.
GWL_USERDATA	Retrieves the 32-bit value associated with the window. Each window has a corresponding 32-bit value intended for use by the application that created the window. This value is initially zero.

The following values are also available when the hWnd parameter identifies a dialog box.

Value	Action
DWL_DLGPROC	Retrieves the address of the dialog box procedure, or a handle representing the address of the dialog box procedure. You must use the CallWindowProc function to call the dialog box procedure.
DWL_MSGRESULT	Retrieves the return value of a message processed in the dialog box procedure.
DWL_USER	Retrieves extra information private to the application, such as handles or pointers.

Return Values

If the function succeeds, the return value is the requested 32-bit value.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

If SetWindowLong has not been called previously, GetWindowLong returns zero for values in the extra window or class memory.

Remarks

Reserve extra window memory by specifying a nonzero value in the cbWndExtra member of the WNDCLASSEX structure used with the RegisterClassEx function.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.246 GetWindowModuleFileName

The GetWindowModuleFileName function retrieves the full path and file name of the module associated with the specified window handle.

```
GetWindowModuleFileName: procedure
(
    hwnd            :dword;
    lpszFileName    :string;
    cchFileNameMax  :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__GetWindowModuleFileName@12" );
```

Parameters

hwnd

[in] Handle to the window whose module file name will be retrieved.

lpszFileName

[out] Pointer to a buffer that receives the path and file name.

cchFileNameMax

[in] Specifies the maximum number of TCHARs that can be copied into the lpszFileName buffer.

Return Values

The return value is the total number of TCHARs copied into the buffer.

Requirements

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.

Windows 95/98: Requires Windows 98 or later.

3.247 GetWindowModuleFileName

The GetWindowModuleFileName function retrieves the full path and file name of the module associated with the specified window handle.

```
GetWindowModuleFileName: procedure
(
    hwnd            :dword;
    lpszFileName    :string;
    cchFileNameMax  :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__GetWindowModuleFileNameA@12" );
```

Parameters

hwnd

[in] Handle to the window whose module file name will be retrieved.

lpszFileName

[out] Pointer to a buffer that receives the path and file name.

cchFileNameMax

[in] Specifies the maximum number of TCHARs that can be copied into the lpszFileName buffer.

Return Values

The return value is the total number of TCHARs copied into the buffer.

Requirements

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.

Windows 95/98: Requires Windows 98 or later.

3.248 GetWindowPlacement

The GetWindowPlacement function retrieves the show state and the restored, minimized, and maximized positions of the specified window.

```
GetWindowPlacement: procedure
(
        hWnd        :dword;
    var lpwndpl     :WINDOWPLACEMENT
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__GetWindowPlacement@8" );
```

Parameters

hWnd

[in] Handle to the window.

lpwndpl

[out] Pointer to the WINDOWPLACEMENT structure that receives the show state and position information.

Before calling GetWindowPlacement, set the length member of the WINDOWPLACEMENT structure to sizeof(WINDOWPLACEMENT). GetWindowPlacement fails if lpwndpl->length is not set correctly.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

The flags member of WINDOWPLACEMENT retrieved by this function is always zero. If the window identified by the hWnd parameter is maximized, the showCmd member is SW_SHOWMAXIMIZED. If the window is minimized, showCmd is SW_SHOWMINIMIZED. Otherwise, it is SW_SHOWNORMAL.

The length member of WINDOWPLACEMENT must be set to sizeof(WINDOWPLACEMENT). If this member is not set correctly, the function returns FALSE. For additional remarks on the proper use of window placement coordinates, see WINDOWPLACEMENT.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.249 GetWindowRect

The GetWindowRect function retrieves the dimensions of the bounding rectangle of the specified window. The dimensions are given in screen coordinates that are relative to the upper-left corner of the screen.

```
GetWindowRect: procedure
(
        hWnd        :dword;
    var lpRect      :RECT
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__GetWindowRect@8" );
```

Parameters

hWnd

[in] Handle to the window.

lpRect

[out] Pointer to a RECT structure that receives the screen coordinates of the upper-left and lower-right corners of the window.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.250 GetWindowRgn

The GetWindowRgn function obtains a copy of the window region of a window. The window region of a window is set by calling the SetWindowRgn function. The window region determines the area within the window where the system permits drawing. The system does not display any portion of a window that lies outside of the window region

```
GetWindowRgn: procedure
(
    hWnd        :dword;
    hRgn        :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__GetWindowRgn@8" );
```

Parameters

hWnd

[in] Handle to the window whose window region is to be obtained.

hRgn

[in] Handle to the region which will be modified to represent the window region.

Return Values

The return value specifies the type of the region that the function obtains. It can be one of the following values.

Value	Meaning
NULLREGION	The region is empty.
SIMPLEREGION	The region is a single rectangle.
COMPLEXREGION	The region is more than one rectangle.
ERROR	The specified window does not have a region, or an error occurred while attempting to return the region.

.

Remarks

The coordinates of a window's window region are relative to the upper-left corner of the window, not the client area of the window.

To set the window region of a window, call the SetWindowRgn function.

The following code shows how you pass in the handle of an existing region.

HRGN hrgn = CreateRectRgn(0,0,0,0);

int regionType = GetWindowRgn(hwnd, hrgn);

if (regionType != ERROR)

{

/* hrgn contains window region */

}

DeleteObject(hrgn); /* finished with region */

Requirements

Windows NT/2000: Requires Windows NT 3.51 or later.

Windows 95/98: Requires Windows 95 or later.

3.251 GetWindowText

The GetWindowText function copies the text of the specified window's title bar (if it has one) into a buffer. If the specified window is a control, the text of the control is copied. However, GetWindowText cannot retrieve the text of a control in another application.

```
GetWindowText: procedure
(
        hWnd        :dword;
    var lpString    :var;
        nMaxCount   :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__GetWindowTextA@12" );
```

Parameters

hWnd

[in] Handle to the window or control containing the text.

lpString

[out] Pointer to the buffer that will receive the text.

nMaxCount

[in] Specifies the maximum number of characters to copy to the buffer, including the NULL character. If the text exceeds this limit, it is truncated.

Return Values

If the function succeeds, the return value is the length, in characters, of the copied string, not including the terminating null character. If the window has no title bar or text, if the title bar is empty, or if the window or control handle is invalid, the return value is zero. To get extended error information, call GetLastError.

This function cannot retrieve the text of an edit control in another application.

Remarks

If the target window is owned by the current process, GetWindowText causes a WM_GETTEXT message to be sent to the specified window or control. If the target window is owned by another process and has a caption, GetWindowText retrieves the window caption text. If the window does not have a caption, the return value is a null string. This behavior is by design. It allows applications to call GetWindowText without hanging if the process that owns the target window is hung. However, if the target window is hung and it belongs to the calling application, GetWindowText will hang the calling application.

To retrieve the text of a control in another process, send a WM_GETTEXT message directly instead of calling GetWindowText.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.252 GetWindowTextLength

The GetWindowTextLength function retrieves the length, in characters, of the specified window's title bar text (if the window has a title bar). If the specified window is a control, the function retrieves the length of the text within the control. However, GetWindowTextLength cannot retrieve the length of the text of an edit control in another application.

```
GetWindowTextLength: procedure
(
    hWnd            :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__GetWindowTextLengthA@4" );
```

Parameters

hWnd

[in] Handle to the window or control.

Return Values

If the function succeeds, the return value is the length, in characters, of the text. Under certain conditions, this value may actually be greater than the length of the text. For more information, see the following Remarks section.

If the window has no text, the return value is zero. To get extended error information, call GetLastError.

Remarks

If the target window is owned by the current process, GetWindowTextLength causes a WM_GETTEXTLENGTH message to be sent to the specified window or control.

Under certain conditions, the GetWindowTextLength function may return a value that is larger than the actual length of the text. This occurs with certain mixtures of ANSI and Unicode, and is due to the system allowing for the possible existence of DBCS characters within the text. The return value, however, will always be at least as large as the actual length of the text; you can thus always use it to guide buffer allocation. This behavior can occur when an application uses both ANSI functions and common dialogs, which use Unicode. It can also occur when an application uses the ANSI version of GetWindowTextLength with a window whose window procedure

is Unicode, or the Unicode version of GetWindowTextLength with a window whose window procedure is ANSI. For more information on ANSI and Unicode functions, see Win32 Function Prototypes.

To obtain the exact length of the text, use the WM_GETTEXT, LB_GETTEXT, or CB_GETLBTEXT messages, or the GetWindowText function.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.253 GetWindowThreadProcessId

The GetWindowThreadProcessId function retrieves the identifier of the thread that created the specified window and, optionally, the identifier of the process that created the window.

```
GetWindowThreadProcessId: procedure
(
        hwnd            :dword;
    var lpdwProcessID   :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__GetWindowThreadProcessId@8" );
```

Parameters

hWnd

[in] Handle to the window.

lpdwProcessId

[out] Pointer to a variable that receives the process identifier. If this parameter is not NULL, GetWindowThreadProcessId copies the identifier of the process to the variable; otherwise, it does not.

Return Values

The return value is the identifier of the thread that created the window.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.254 GrayString

The GrayString function draws gray text at the specified location. The function draws the text by copying it into a memory bitmap, graying the bitmap, and then copying the bitmap to the screen. The function grays the text regardless of the selected brush and background. GrayString uses the font currently selected for the specified device context.

If the lpOutputFunc parameter is NULL, GDI uses the TextOut function, and the lpData parameter is assumed to be a pointer to the character string to be output. If the characters to be output cannot be handled by TextOut (for example, the string is stored as a bitmap), the application must supply its own output function.

```
GrayString: procedure
(
```

```
    hDC             :dword;
    hBrush          :dword;
    lpOutputFunc    :GRAYSTRINGPROC;
    _lpData         :dword;
    nCount          :dword;
    X               :dword;
    Y               :dword;
    nWidth          :dword;
    nHeight         :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__GrayStringA@36" );
```

Parameters

hDC

[in] Handle to the device context.

hBrush

[in] Handle to the brush to be used for graying. If this parameter is NULL, the text is grayed with the same brush that was used to draw window text.

lpOutputFunc

[in] Pointer to the application-defined function that will draw the string, or, if TextOut is to be used to draw the string, it is a NULL pointer. For details, see the OutputProc callback function.

lpData

[in] Specifies a pointer to data to be passed to the output function. If the lpOutputFunc parameter is NULL, lpData must be a pointer to the string to be output.

nCount

[in] Specifies the number of characters to be output. If the nCount parameter is zero, GrayString calculates the length of the string (assuming lpData is a pointer to the string). If nCount is –1 and the function pointed to by lpOutputFunc returns FALSE, the image is shown but not grayed.

X

[in] Specifies the device x-coordinate of the starting position of the rectangle that encloses the string.

Y

[in] Specifies the device y-coordinate of the starting position of the rectangle that encloses the string.

nWidth

[in] Specifies the width, in device units, of the rectangle that encloses the string. If this parameter is zero, GrayString calculates the width of the area, assuming lpData is a pointer to the string.

nHeight

[in] Specifies the height, in device units, of the rectangle that encloses the string. If this parameter is zero, GrayString calculates the height of the area, assuming lpData is a pointer to the string.

Return Values

If the string is drawn, the return value is nonzero.

If either the TextOut function or the application-defined output function returned zero, or there was insufficient memory to create a memory bitmap for graying, the return value is zero.

Windows NT/ 2000: To get extended error information, call GetLastError.

Remarks

Without calling GrayString, an application can draw grayed strings on devices that support a solid gray color. The system color COLOR_GRAYTEXT is the solid-gray system color used to draw disabled text. The application can call the GetSysColor function to retrieve the color value of COLOR_GRAYTEXT. If the color is other than zero (black), the application can call the SetTextColor function to set the text color to the color value and then draw the string directly. If the retrieved color is black, the application must call GrayString to gray the text.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.255 HideCaret

The HideCaret function removes the caret from the screen. Hiding a caret does not destroy its current shape or invalidate the insertion point.

```
HideCaret: procedure
(
    hwnd            :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__HideCaret@4" );
```

Parameters

hWnd

[in] Handle to the window that owns the caret. If this parameter is NULL, HideCaret searches the current task for the window that owns the caret.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

HideCaret hides the caret only if the specified window owns the caret. If the specified window does not own the caret, HideCaret does nothing and returns FALSE.

Hiding is cumulative. If your application calls HideCaret five times in a row, it must also call ShowCaret five times before the caret is displayed.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.256 HiliteMenuItem

The HiliteMenuItem function highlights or removes the highlighting from an item in a menu bar.

```
HiliteMenuItem: procedure
(
```

```
    hwnd            :dword;
    hmenu           :dword;
    uItemHilite     :dword;
    uHilite         :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__HiliteMenuItem@16" );
```

Parameters

hwnd

[in] Handle to the window that contains the menu.

hmenu

[in] Handle to the menu bar that contains the item to be highlighted.

uItemHilite

[in] Specifies the menu item to be highlighted. This parameter is either the identifier of the menu item or the offset of the menu item in the menu bar, depending on the value of the uHilite parameter.

uHilite

[in] Controls the interpretation of the uItemHilite parameter and indicates whether the menu item is highlighted. This parameter must be a combination of either MF_BYCOMMAND or MF_BYPOSITION and MF_HILITE or MF_UNHILITE.

Value	Meaning
MF_BYCOMMAND	Indicates that uItemHilite gives the identifier of the menu item.
MF_BYPOSITION	Indicates that uItemHilite gives the zero-based relative position of the menu item.
MF_HILITE	Highlights the menu item. If this flag is not specified, the highlighting is removed from the item.
MF_UNHILITE	Removes highlighting from the menu item.

Return Values

If the menu item is set to the specified highlight state, the return value is nonzero.

If the menu item is not set to the specified highlight state, the return value is zero.

Remarks

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

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.257 ImpersonateDdeClientWindow

The ImpersonateDdeClientWindow function enables a DDE server application to impersonate a DDE client application's security context. This protects secure server data from unauthorized DDE clients.

```
ImpersonateDdeClientWindow: procedure
(
    hWndClient  :dword;
```

```
    hWndServer  :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__ImpersonateDdeClientWindow@8" );
```

Parameters

hWndClient

[in] Handle to the DDE client window to impersonate. The client window must have established a DDE conversation with the server window identified by the hWndServer parameter.

hWndServer

[in] Handle to the DDE server window. An application must create the server window before calling this function.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

An application should call the RevertToSelf function to undo the impersonation set by the ImpersonateDdeClientWindow function.

A DDEML application should use the DdeImpersonateClient function.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Unsupported.

3.258 InSendMessage

The InSendMessage function determines whether the current window procedure is processing a message that was sent from another thread (in the same process or a different process) by a call to the SendMessage function.

To obtain additional information about how the message was sent, use the InSendMessageEx function.

```
InSendMessage: procedure;
    @stdcall;
    @returns( "eax" );
    @external( "__imp__InSendMessage@0" );
```

Parameters

This function has no parameters.

Return Values

If the window procedure is processing a message sent to it from another thread using the SendMessage function, the return value is nonzero.

If the window procedure is not processing a message sent to it from another thread using the SendMessage function, the return value is zero.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.259 InSendMessageEx

The InSendMessageEx function determines whether the current window procedure is processing a message that was sent from another thread (in the same process or a different process).

```
InSendMessageEx: procedure
(
    var lpReserved  :var
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__InSendMessageEx@4" );
```

Parameters

lpReserved

Reserved; must be NULL.

Return Values

If the message was not sent, the return value is ISMEX_NOSEND. Otherwise, the return value is one or more of the following values.

Value	Meaning
ISMEX_CALLBACK	The message was sent using the SendMessageCallback function. The thread that sent the message is not blocked.
ISMEX_NOTIFY	The message was sent using the SendNotifyMessage function. The thread that sent the message is not blocked.
ISMEX_REPLIED	The window procedure has processed the message. The thread that sent the message is no longer blocked.
ISMEX_SEND	The message was sent using the SendMessage or SendMessageTimeout function. If ISMEX_REPLIED is not set, the thread that sent the message is blocked.

Remarks

To determine if the sender is blocked, use the following test:

fBlocked = (InSendMessageEx(NULL) & (ISMEX_REPLIED|ISMEX_SEND)) == ISMEX_SEND;

Requirements

Windows NT/2000: Requires Windows 2000 or later.

Windows 95/98: Requires Windows 98 or later.

3.260 InflateRect

The InflateRect function increases or decreases the width and height of the specified rectangle. The InflateRect function adds dx units to the left and right ends of the rectangle and dy units to the top and bottom. The dx and dy parameters are signed values; positive values increase the width and height, and negative values decrease them.

```
InflateRect: procedure
```

```
(
    var lprc    :RECT;
        dx      :dword;
        dy      :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__InflateRect@12" );
```

Parameters

lprc

[in/out] Pointer to the RECT structure that increases or decreases in size.

dx

[in] Specifies the amount to increase or decrease the rectangle width. This parameter must be negative to decrease the width.

dy

[in] Specifies the amount to increase or decrease the rectangle height. This parameter must be negative to decrease the height.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Windows NT/ 2000: To get extended error information, call GetLastError.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.261 InsertMenu

The InsertMenu function inserts a new menu item into a menu, moving other items down the menu.

Note The InsertMenu function has been superseded by the InsertMenuItem function. You can still use Insert-Menu, however, if you do not need any of the extended features of InsertMenuItem.

```
InsertMenu: procedure
(
        hMenu       :dword;
        uPosition   :dword;
        uFlags      :dword;
        uIDNewItem  :dword;
        lpNewItem   :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__InsertMenuA@20" );
```

Parameters

hMenu

[in] Handle to the menu to be changed.

uPosition

[in] Specifies the menu item before which the new menu item is to be inserted, as determined by the uFlags parameter.

uFlags

[in] Specifies flags that control the interpretation of the uPosition parameter and the content, appearance, and behavior of the new menu item. This parameter must be a combination of one of the following required values and at least one of the values listed in the following Remarks section.

Value	Description
MF_BYCOMMAND	Indicates that the uPosition parameter gives the identifier of the menu item. The MF_BYCOMMAND flag is the default if neither the MF_BYCOMMAND nor MF_BYPOSITION flag is specified.
MF_BYPOSITION	Indicates that the uPosition parameter gives the zero-based relative position of the new menu item. If uPosition is -1, the new menu item is appended to the end of the menu.

uIDNewItem

[in] Specifies either the identifier of the new menu item or, if the uFlags parameter has the MF_POPUP flag set, a handle to the drop-down menu or submenu.

lpNewItem

[in] Specifies the content of the new menu item. The interpretation of lpNewItem depends on whether the uFlags parameter includes the MF_BITMAP, MF_OWNERDRAW, or MF_STRING flag, as follows.

Value	Description
MF_BITMAP	Contains a bitmap handle.
MF_OWNERDRAW	Contains an application-supplied value that can be used to maintain additional data related to the menu item. The value is in the itemData member of the structure pointed to by the lparam parameter of the WM_MEASUREITEM or WM_DRAWITEM message sent when the menu item is created or its appearance is updated.
MF_STRING	Contains a pointer to a null-terminated string (the default).

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

The application must call the DrawMenuBar function whenever a menu changes, whether or not the menu is in a displayed window.

The following list describes the flags that can be set in the uFlags parameter.

Value	Description
MF_BITMAP	Uses a bitmap as the menu item. The lpNewItem parameter contains a handle to the bitmap.
MF_CHECKED	Places a check mark next to the menu item. If the application provides check-mark bitmaps (see SetMenuItemBitmaps), this flag displays the check-mark bitmap next to the menu item.
MF_DISABLED	Disables the menu item so that it cannot be selected, but does not gray it.
MF_ENABLED	Enables the menu item so that it can be selected and restores it from its grayed state.
MF_GRAYED	Disables the menu item and grays it so it cannot be selected.

MF_MENUBARBREAK	Functions the same as the MF_MENUBREAK flag for a menu bar. For a drop-down menu, submenu, or shortcut menu, the new column is separated from the old column by a vertical line.
MF_MENUBREAK	Places the item on a new line (for menu bars) or in a new column (for a drop-down menu, submenu, or shortcut menu) without separating columns.
MF_OWNERDRAW	Specifies that the item is an owner-drawn item. Before the menu is displayed for the first time, the window that owns the menu receives a WM_MEASUREITEM message to retrieve the width and height of the menu item. The WM_DRAWITEM message is then sent to the window procedure of the owner window whenever the appearance of the menu item must be updated.
MF_POPUP	Specifies that the menu item opens a drop-down menu or submenu. The uIDNewItem parameter specifies a handle to the drop-down menu or submenu. This flag is used to add a menu name to a menu bar or a menu item that opens a submenu to a drop-down menu, submenu, or shortcut menu.
MF_SEPARATOR	Draws a horizontal dividing line. This flag is used only in a drop-down menu, submenu, or shortcut menu. The line cannot be grayed, disabled, or highlighted. The lpNewItem and uIDNewItem parameters are ignored.
MF_STRING	Specifies that the menu item is a text string; the lpNewItem parameter is a pointer to the string.
MF_UNCHECKED	Does not place a check mark next to the menu item (default). If the application supplies check-mark bitmaps (see the SetMenuItemBitmaps function), this flag displays the clear bitmap next to the menu item.

The following groups of flags cannot be used together:

- MF_BYCOMMAND and MF_BYPOSITION
- MF_DISABLED, MF_ENABLED, and MF_GRAYED
- MF_BITMAP, MF_STRING, MF_OWNERDRAW, and MF_SEPARATOR
- MF_MENUBARBREAK and MF_MENUBREAK
- MF_CHECKED and MF_UNCHECKED

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.262 InsertMenuItem

The InsertMenuItem function inserts a new menu item at the specified position in a menu.

```
InsertMenuItem: procedure
(
        hMenu           :dword;
        uItem           :dword;
        fByPosition     :boolean;
    var lpmii           :CMENUITEMINFO
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__InsertMenuItemA@16" );
```

Parameters

hMenu

[in] Handle to the menu in which the new menu item is inserted.

uItem

[in] Identifier or position of the menu item before which to insert the new item. The meaning of this parameter depends on the value of fByPosition.

fByPosition

[in] Value specifying the meaning of uItem. If this parameter is FALSE, uItem is a menu item identifier. Otherwise, it is a menu item position.

lpmii

[in] Pointer to a MENUITEMINFO structure that contains information about the new menu item.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, use the GetLastError function.

Remarks

The application must call the DrawMenuBar function whenever a menu changes, whether or not the menu is in a displayed window.

In order for keyboard accelerators to work with bitmap or owner-drawn menu items, the owner of the menu must process the WM_MENUCHAR message. See Owner-Drawn Menus and the WM_MENUCHAR Message for more information.

Requirements

Windows NT/2000: Requires Windows NT 4.0 or later.

Windows 95/98: Requires Windows 95 or later.

3.263 IntersectRect

The IntersectRect function calculates the intersection of two source rectangles and places the coordinates of the intersection rectangle into the destination rectangle. If the source rectangles do not intersect, an empty rectangle (in which all coordinates are set to zero) is placed into the destination rectangle.

```
IntersectRect: procedure
(
    var lprcDst     :RECT;
    var lprcSrc1    :RECT;
    var lprcSrc2    :RECT
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__IntersectRect@12" );
```

Parameters

lprcDst

[out] Pointer to the RECT structure that is to receive the intersection of the rectangles pointed to by the lprcSrc1

and lprcSrc2 parameters. This parameter cannot be NULL.

lprcSrc1

[in] Pointer to the RECT structure that contains the first source rectangle.

lprcSrc2

[in] Pointer to the RECT structure that contains the second source rectangle.

Return Values

If the rectangles intersect, the return value is nonzero.

If the rectangles do not intersect, the return value is zero.

Windows NT/ 2000: To get extended error information, call GetLastError.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.264 InvalidateRect

The InvalidateRect function adds a rectangle to the specified window's update region. The update region represents the portion of the window's client area that must be redrawn.

```
InvalidateRect: procedure
(
        hWnd        :dword;
    var lpRect      :RECT;
        bErase      :boolean
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__InvalidateRect@12" );
```

Parameters

hWnd

[in] Handle to the window whose update region has changed. If this parameter is NULL, the system invalidates and redraws all windows, and sends the WM_ERASEBKGND and WM_NCPAINT messages to the window procedure before the function returns.

lpRect

[in] Pointer to a RECT structure that contains the client coordinates of the rectangle to be added to the update region. If this parameter is NULL, the entire client area is added to the update region.

bErase

[in] Specifies whether the background within the update region is to be erased when the update region is processed. If this parameter is TRUE, the background is erased when the BeginPaint function is called. If this parameter is FALSE, the background remains unchanged.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Windows NT/ 2000: To get extended error information, call GetLastError.

Remarks

The invalidated areas accumulate in the update region until the region is processed when the next WM_PAINT message occurs or until the region is validated by using the ValidateRect or ValidateRgn function.

The system sends a WM_PAINT message to a window whenever its update region is not empty and there are no other messages in the application queue for that window.

If the bErase parameter is TRUE for any part of the update region, the background is erased in the entire region, not just in the specified part.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.265 InvalidateRgn

The InvalidateRgn function invalidates the client area within the specified region by adding it to the current update region of a window. The invalidated region, along with all other areas in the update region, is marked for painting when the next WM_PAINT message occurs.

```
InvalidateRgn: procedure
(
    hWnd        :dword;
    hRgn        :dword;
    bErase      :boolean
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__InvalidateRgn@12" );
```

Parameters

hWnd

[in] Handle to the window with an update region that is to be modified.

hRgn

[in] Handle to the region to be added to the update region. The region is assumed to have client coordinates. If this parameter is NULL, the entire client area is added to the update region.

bErase

[in] Specifies whether the background within the update region should be erased when the update region is processed. If this parameter is TRUE, the background is erased when the BeginPaint function is called. If the parameter is FALSE, the background remains unchanged.

Return Values

The return value is always nonzero.

Remarks

Invalidated areas accumulate in the update region until the next WM_PAINT message is processed or until the region is validated by using the ValidateRect or ValidateRgn function.

The system sends a WM_PAINT message to a window whenever its update region is not empty and there are no other messages in the application queue for that window.

The specified region must have been created by using one of the region functions.

If the bErase parameter is TRUE for any part of the update region, the background in the entire region is erased, not just in the specified part.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.266 InvertRect

The InvertRect function inverts a rectangle in a window by performing a logical NOT operation on the color values for each pixel in the rectangle's interior.

```
InvertRect: procedure
(
        hDC     :dword;
    var lprc    :RECT
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__InvertRect@8" );
```

Parameters

hDC

[in] Handle to the device context.

lprc

[in] Pointer to a RECT structure that contains the logical coordinates of the rectangle to be inverted.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Windows NT/ 2000: To get extended error information, call GetLastError.

Remarks

On monochrome screens, InvertRect makes white pixels black and black pixels white. On color screens, the inversion depends on how colors are generated for the screen. Calling InvertRect twice for the same rectangle restores the display to its previous colors.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.267 IsCharAlpha

The IsCharAlpha function determines whether a character is an alphabetic character. This determination is based on the semantics of the language selected by the user during setup or through Control Panel.

```
IsCharAlpha: procedure
(
    _ch         :char
```

```
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__IsCharAlphaA@4" );
```

Parameters

ch

[in] Specifies the character to be tested.

Return Values

If the character is alphabetic, the return value is nonzero.

If the character is not alphabetic, the return value is zero. To get extended error information, call GetLastError.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.268 IsCharAlphaNumeric

The IsCharAlphaNumeric function determines whether a character is either an alphabetic or a numeric character. This determination is based on the semantics of the language selected by the user during setup or through Control Panel.

```
IsCharAlphaNumeric: procedure
(
    _ch         :char
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__IsCharAlphaNumericA@4" );
```

Parameters

ch

[in] Specifies the character to be tested.

Return Values

If the character is alphanumeric, the return value is nonzero.

If the character is not alphanumeric, the return value is zero. To get extended error information, call GetLastError.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.269 IsCharLower

The IsCharLower function determines whether a character is lowercase. This determination is based on the semantics of the language selected by the user during setup or through Control Panel.

```
IsCharLower: procedure
(
    _ch         :char
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__IsCharLowerA@4" );
```

Parameters

ch

[in] Specifies the character to be tested.

Return Values

If the character is lowercase, the return value is nonzero.

If the character is not lowercase, the return value is zero. To get extended error information, call GetLastError.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.270 IsCharUpper

The IsCharUpper function determines whether a character is uppercase. This determination is based on the semantics of the language selected by the user during setup or through Control Panel.

```
IsCharUpper: procedure
(
    _ch         :char
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__IsCharUpperA@4" );
```

Parameters

ch

[in] Specifies the character to be tested.

Return Values

If the character is uppercase, the return value is nonzero.

If the character is not uppercase, the return value is zero. To get extended error information, call GetLastError.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.271 IsChild

The IsChild function tests whether a window is a child window or descendant window of a specified parent win-

dow. A child window is the direct descendant of a specified parent window if that parent window is in the chain of parent windows; the chain of parent windows leads from the original overlapped or pop-up window to the child window.

```
IsChild: procedure
(
    hWndParent      :dword;
    hWnd            :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__IsChild@8" );
```

Parameters

hWndParent

[in] Handle to the parent window.

hWnd

[in] Handle to the window to be tested.

Return Values

If the window is a child or descendant window of the specified parent window, the return value is nonzero.

If the window is not a child or descendant window of the specified parent window, the return value is zero.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.272 IsClipboardFormatAvailable

The IsClipboardFormatAvailable function determines whether the clipboard contains data in the specified format.

```
IsClipboardFormatAvailable: procedure
(
    format          :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__IsClipboardFormatAvailable@4" );
```

Parameters

format

[in] Specifies a standard or registered clipboard format. For a description of the standard clipboard formats, see Standard Clipboard Formats.

Return Values

If the clipboard format is available, the return value is nonzero.

If the clipboard format is not available, the return value is zero. To get extended error information, call GetLastError.

Remarks

Typically, an application that recognizes only one clipboard format would call this function when processing the WM_INITMENU or WM_INITMENUPOPUP message. The application would then enable or disable the Paste menu item, depending on the return value. Applications that recognize more than one clipboard format should use the GetPriorityClipboardFormat function for this purpose.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.273 IsDialogMessage

The IsDialogMessage function determines whether a message is intended for the specified dialog box and, if it is, processes the message.

```
IsDialogMessage: procedure
(
        hDlg        :dword;
    var lpMsg       :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__IsDialogMessageA@8" );

IsDialogMessageW: procedure
(
        hDlg        :dword;
    var lpMsg       :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__IsDialogMessage@8" );
```

Parameters

hDlg

[in] Handle to the dialog box.

lpMsg

[in] Pointer to an MSG structure that contains the message to be checked.

Return Values

If the message has been processed, the return value is nonzero.

If the message has not been processed, the return value is zero.

Remarks

Although the IsDialogMessage function is intended for modeless dialog boxes, you can use it with any window that contains controls, enabling the windows to provide the same keyboard selection as is used in a dialog box.

When IsDialogMessage processes a message, it checks for keyboard messages and converts them into selections for the corresponding dialog box. For example, the TAB key, when pressed, selects the next control or group of controls, and the DOWN ARROW key, when pressed, selects the next control in a group.

Because the IsDialogMessage function performs all necessary translating and dispatching of messages, a message processed by IsDialogMessage must not be passed to the TranslateMessage or DispatchMessage function.

IsDialogMessage sends WM_GETDLGCODE messages to the dialog box procedure to determine which keys should be processed.

IsDialogMessage can send DM_GETDEFID and DM_SETDEFID messages to the window. These messages are defined in the WINUSER.H header file as WM_USER and WM_USER + 1, so conflicts are possible with application-defined messages having the same values.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.274 IsDlgButtonChecked

The IsDlgButtonChecked function determines whether a button control has a check mark next to it or whether a three-state button control is grayed, checked, or neither.

```
IsDlgButtonChecked: procedure
(
    hDlg            :dword;
    nIDButton       :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__IsDlgButtonChecked@8" );
```

Parameters

hDlg

[in] Handle to the dialog box that contains the button control.

nIDButton

[in] Specifies the identifier of the button control.

Return Values

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

Value	Meaning
BST_CHECKED	Button is checked.
BST_INDETERMINATE	Button is grayed, indicating an indeterminate state (applies only if the button has the BS_3STATE or BS_AUTO3STATE style).
BST_UNCHECKED	Button is cleared

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

Remarks

The IsDlgButtonChecked function sends a BM_GETCHECK message to the specified button control.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.275 IsIconic

The IsIconic function determines whether the specified window is minimized (iconic).

```
IsIconic: procedure
(
    hWnd            :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__IsIconic@4" );
```

Parameters

hWnd

[in] Handle to the window to test.

Return Values

If the window is iconic, the return value is nonzero.

If the window is not iconic, the return value is zero.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.276 IsMenu

The IsMenu function determines whether a handle is a menu handle.

```
IsMenu: procedure
(
    hMenu           :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__IsMenu@4" );
```

Parameters

hMenu

[in] Handle to be tested.

Return Values

If hMenu is a menu handle, the return value is nonzero.

If hMenu is not a menu handle, the return value is zero.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.277 IsRectEmpty

The IsRectEmpty function determines whether the specified rectangle is empty. A empty rectangle is one that has no area; that is, the coordinate of the right side is less than or equal to the coordinate of the left side, or the coordinate of the bottom side is less than or equal to the coordinate of the top side.

```
IsRectEmpty: procedure
(
    var lprc    :RECT
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__IsRectEmpty@4" );
```

Parameters

lprc

[in] Pointer to a RECT structure that contains the logical coordinates of the rectangle.

Return Values

If the rectangle is empty, the return value is nonzero.

If the rectangle is not empty, the return value is zero.

Windows NT/ 2000: To get extended error information, call GetLastError.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.278 IsWindow

The IsWindow function determines whether the specified window handle identifies an existing window.

```
IsWindow: procedure
(
    hWnd        :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__IsWindow@4" );
```

Parameters

hWnd

[in] Handle to the window to test.

Return Values

If the window handle identifies an existing window, the return value is nonzero.

If the window handle does not identify an existing window, the return value is zero.

Remarks

An application should not use IsWindow for a window that it did not create because the window could be destroyed after this function was called. Further, because window handles are recycled the handle could even

point to a different window.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.279 IsWindowEnabled

The IsWindowEnabled function determines whether the specified window is enabled for mouse and keyboard input.

```
IsWindowEnabled: procedure
(
    hWnd        :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__IsWindowEnabled@4" );
```

Parameters

hWnd

[in] Handle to the window to test.

Return Values

If the window is enabled, the return value is nonzero.

If the window is not enabled, the return value is zero.

Remarks

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

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.280 IsWindowUnicode

The IsWindowUnicode function determines whether the specified window is a native Unicode window.

```
IsWindowUnicode: procedure
(
    hWnd        :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__IsWindowUnicode@4" );
```

Parameters

hWnd

[in] Handle to the window to test.

Return Values

If the window is a native Unicode window, the return value is nonzero.

If the window is not a native Unicode window, the return value is zero. The window is a native ANSI window.

Remarks

The character set of a window is determined by the use of the RegisterClass function. If the window class was registered with the ANSI version of RegisterClass (RegisterClassA), the character set of the window is ANSI. If the window class was registered with the Unicode version of RegisterClass (RegisterClassW), the character set of the window is Unicode.

The system does automatic two-way translation (Unicode to ANSI) for window messages. For example, if an ANSI window message is sent to a window that uses the Unicode character set, the system translates that message into a Unicode message before calling the window procedure. The system calls IsWindowUnicode to determine whether to translate the message.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.281 IsWindowVisible

The IsWindowVisible function retrieves the visibility state of the specified window.

```
IsWindowVisible: procedure
(
    hWnd        :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__IsWindowVisible@4" );
```

Parameters

hWnd

[in] Handle to the window to test.

Return Values

If the specified window, its parent window, its parent's parent window, and so forth, have the WS_VISIBLE style, the return value is nonzero. Otherwise, the return value is zero.

Because the return value specifies whether the window has the WS_VISIBLE style, it may be nonzero even if the window is totally obscured by other windows.

Remarks

The visibility state of a window is indicated by the WS_VISIBLE style bit. When WS_VISIBLE is set, the window is displayed and subsequent drawing into it is displayed as long as the window has the WS_VISIBLE style.

Any drawing to a window with the WS_VISIBLE style will not be displayed if the window is obscured by other windows or is clipped by its parent window.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.282 IsZoomed

The IsZoomed function determines whether a window is maximized.

```
IsZoomed: procedure
(
    hWnd        :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__IsZoomed@4" );
```

Parameters

hWnd

[in] Handle to the window to test.

Return Values

If the window is zoomed, the return value is nonzero.

If the window is not zoomed, the return value is zero.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.283 keybd_event

The keybd_event function synthesizes a keystroke. The system can use such a synthesized keystroke to generate a WM_KEYUP or WM_KEYDOWN message. The keyboard driver's interrupt handler calls the keybd_event function.

Windows NT/2000: This function has been superseded. Use SendInput instead.

```
keybd_event: procedure
(
    bVK         :byte;
    bScan       :byte;
    dwFlags     :dword;
    dwExtraInfo :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__keybd_event@16" );
```

Parameters

bVk

[in] Specifies a virtual-key code. The code must be a value in the range 1 to 254. For a complete list, see Virtual-Key Codes.

bScan

This parameter is not used.

dwFlags

[in] Specifies various aspects of function operation. This parameter can be one or more of the following values.

Value	Meaning
KEYEVENTF_EXTENDEDKEY	If specified, the scan code was preceded by a prefix byte having the value 0xE0 (224).
KEYEVENTF_KEYUP	If specified, the key is being released. If not specified, the key is being depressed.

dwExtraInfo

[in] Specifies an additional value associated with the key stroke.

Return Values

This function has no return value.

Remarks

An application can simulate a press of the PRINTSCRN key in order to obtain a screen snapshot and save it to the clipboard. To do this, call keybd_event with the bVk parameter set to VK_SNAPSHOT.

Windows NT: The keybd_event function can toggle the NUM LOCK, CAPS LOCK, and SCROLL LOCK keys.

Windows 95: The keybd_event function can toggle only the CAPS LOCK and SCROLL LOCK keys. It cannot toggle the NUM LOCK key.

The following sample program toggles the NUM LOCK light (on Windows NT only) by using keybd_event() with a virtual key of VK_NUMLOCK. It takes a Boolean value that indicates whether the light should be turned off (FALSE) or on (TRUE). The same technique can be used for the CAPS LOCK key (VK_CAPITAL) and the SCROLL LOCK key (VK_SCROLL).

```
#include <windows.h>

void SetNumLock( BOOL bState )
{
   BYTE keyState[256];

   GetKeyboardState((LPBYTE)&keyState);
   if( (bState && !(keyState[VK_NUMLOCK] & 1)) ||
       (!bState && (keyState[VK_NUMLOCK] & 1)) )
   {
   // Simulate a key press
      keybd_event( VK_NUMLOCK,
                   0x45,
                   KEYEVENTF_EXTENDEDKEY | 0,
                   0 );

   // Simulate a key release
      keybd_event( VK_NUMLOCK,
                   0x45,
                   KEYEVENTF_EXTENDEDKEY | KEYEVENTF_KEYUP,
                   0);
   }
}

void main()
{
   SetNumLock( TRUE );
}
```

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.284 KillTimer

The KillTimer function destroys the specified timer.

```
KillTimer: procedure
(
    hWnd        :dword;
    uIDEvent    :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__KillTimer@8" );
```

Parameters

hWnd

[in] Handle to the window associated with the specified timer. This value must be the same as the hWnd value passed to the SetTimer function that created the timer.

uIDEvent

[in] Specifies the timer to be destroyed. If the window handle passed to SetTimer is valid, this parameter must be the same as the uIDEvent value passed to SetTimer. If the application calls SetTimer with hWnd set to NULL, this parameter must be the timer identifier returned by SetTimer.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

The KillTimer function does not remove WM_TIMER messages already posted to the message queue.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.285 LoadAccelerators

The LoadAccelerators function loads the specified accelerator table.

```
LoadAccelerators: procedure
(
    hInstance   :dword;
    lpTableName :string
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__LoadAcceleratorsA@8" );
```

Parameters

hInstance

[in] Handle to the module whose executable file contains the accelerator table to load.

lpTableName

[in] Pointer to a null-terminated string that contains the name of the accelerator table to load. Alternatively, this parameter can specify the resource identifier of an accelerator-table resource in the low-order word and zero in the high-order word. To create this value, use the MAKEINTRESOURCE macro.

Return Values

If the function succeeds, the return value is a handle to the loaded accelerator table.

If the function fails, the return value is NULL. To get extended error information, call GetLastError.

Remarks

If the accelerator table has not yet been loaded, the function loads it from the specified executable file.

Accelerator tables loaded from resources are freed automatically when the application terminates.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.286 LoadBitmap

The LoadBitmap function loads the specified bitmap resource from a module's executable file. This function has been superseded by the LoadImage function.

```
LoadBitmap: procedure
(
    hInstance       :dword;
    lpBitmapName    :string
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__LoadBitmapA@8" );
```

Parameters

hInstance

[in] Handle to the instance of the module whose executable file contains the bitmap to be loaded.

lpBitmapName

[in] Pointer to a null-terminated string that contains the name of the bitmap resource to be loaded. Alternatively, this parameter can consist of the resource identifier in the low-order word and zero in the high-order word. The MAKEINTRESOURCE macro can be used to create this value.

Return Values

If the function succeeds, the return value is the handle to the specified bitmap.

If the function fails, the return value is NULL.

Windows NT/ 2000: To get extended error information, call GetLastError.

Remarks

If the bitmap pointed to by the lpBitmapName parameter does not exist or there is insufficient memory to load the bitmap, the function fails.

LoadBitmap creates a compatible bitmap of the display, which cannot be selected to a printer. To load a bitmap that you can select to a printer, call LoadImage and specify LR_CREATEDIBSECTION to create a DIB section. A DIB section can be selected to any device.

An application can use the LoadBitmap function to access the predefined bitmaps used by the Win32 API. To do so, the application must set the hInstance parameter to NULL and the lpBitmapName parameter to one of the following values.

Bitmap name	Bitmap name
OBM_BTNCORNERS	OBM_OLD_RESTORE
OBM_BTSIZE	OBM_OLD_RGARROW
OBM_CHECK	OBM_OLD_UPARROW
OBM_CHECKBOXES	OBM_OLD_ZOOM
OBM_CLOSE	OBM_REDUCE
OBM_COMBO	OBM_REDUCED
OBM_DNARROW	OBM_RESTORE
OBM_DNARROWD	OBM_RESTORED
OBM_DNARROWI	OBM_RGARROW
OBM_LFARROW	OBM_RGARROWD
OBM_LFARROWD	OBM_RGARROWI
OBM_LFARROWI	OBM_SIZE
OBM_MNARROW	OBM_UPARROW
OBM_OLD_CLOSE	OBM_UPARROWD
OBM_OLD_DNARROW	OBM_UPARROWI
OBM_OLD_LFARROW	OBM_ZOOM
OBM_OLD_REDUCE	OBM_ZOOMD

Bitmap names that begin with OBM_OLD represent bitmaps used by 16-bit versions of Windows earlier than 3.0.

For an application to use any of the OBM_ constants, the constant OEMRESOURCE must be defined before the Windows.h header file is included.

The application must call the DeleteObject function to delete each bitmap handle returned by the LoadBitmap function.

Windows 95 has a problem dealing with Win32 .exe or .dll files that contain resources whose size is 64K or larger. To retain Win16 compatibility, Windows 95 converts the 32-bit size into a 16-bit size and a shift count. When it does this conversion it rounds down instead of up, so some bytes can be lost. In addition, Win16 uses the same shift count for all resources, thus the shift required for a large resource can cause a small resource to be severely truncated, or even eliminated completely.

To avoid this problem, compute the scaling factor for the largest resource and pad all resources with zeroes so each is a multiple of the scaling factor. For example, a resource of size 0x100065 is converted to 0x8003 * 32, which loses 5 bytes. To save the 5 bytes, you must pad the resource with 27 zeroes so that it becomes size 0x100080 and is then converted to 0x8004 * 32. Note, any smaller resource must also be padded with zeroes so it is a multiple of the scaling factor, which in this case is 32.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.287 LoadCursor

The LoadCursor function loads the specified cursor resource from the executable (.EXE) file associated with an application instance.

Note This function has been superseded by the LoadImage function.

```
LoadCursor: procedure
(
    hInstance       :dword;
    lpCursorName    :string
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__LoadCursorA@8" );
```

Parameters

hInstance

[in] Handle to an instance of the module whose executable file contains the cursor to be loaded.

lpCursorName

[in] Pointer to a null-terminated string that contains the name of the cursor resource to be loaded. Alternatively, this parameter can consist of the resource identifier in the low-order word and zero in the high-order word. The MAKEINTRESOURCE macro can also be used to create this value.

To use one of the cursors predefined in the Microsoft® Win32® API, the application must set the hInstance parameter to NULL and the lpCursorName parameter to one the following values:

Value	Meaning
IDC_APPSTARTING	Standard arrow and small hourglass
IDC_ARROW	Standard arrow
IDC_CROSS	Crosshair
IDC_HAND	Windows 2000: Hand
IDC_HELP	Arrow and question mark
IDC_IBEAM	I-beam
IDC_ICON	Obsolete for applications marked version 4.0 or later.
IDC_NO	Slashed circle
IDC_SIZE	Obsolete for applications marked version 4.0 or later. Use IDC_SIZEALL.
IDC_SIZEALL	Four-pointed arrow pointing north, south, east, and west
IDC_SIZENESW	Double-pointed arrow pointing northeast and southwest
IDC_SIZENS	Double-pointed arrow pointing north and south
IDC_SIZENWSE	Double-pointed arrow pointing northwest and southeast
IDC_SIZEWE	Double-pointed arrow pointing west and east
IDC_UPARROW	Vertical arrow
IDC_WAIT	Hourglass

Return Values

If the function succeeds, the return value is the handle to the newly loaded cursor.

If the function fails, the return value is NULL. To get extended error information, call GetLastError.

Remarks

The LoadCursor function loads the cursor resource only if it has not been loaded; otherwise, it retrieves the handle to the existing resource. This function returns a valid cursor handle only if the lpCursorName parameter is a pointer to a cursor resource. If lpCursorName is a pointer to any type of resource other than a cursor (such as an icon), the return value is not NULL, even though it is not a valid cursor handle.

The LoadCursor function searches the cursor resource most appropriate for the cursor for the current display device. The cursor resource can be a color or monochrome bitmap.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.288 LoadCursorFromFile

The LoadCursorFromFile function creates a cursor based on data contained in a file. The file is specified by its name or by a system cursor identifier. The function returns a handle to the newly created cursor. Files containing cursor data may be in either cursor (.CUR) or animated cursor (.ANI) format.

```
LoadCursorFromFile: procedure
(
    lpFileName      :string
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__LoadCursorFromFileA@4" );
```

Parameters

lpFileName

[in] Specifies the source of the file data to be used to create the cursor. The data in the file must be in either .CUR or .ANI format.

If the high-order word of lpszFileName is nonzero, it is a pointer to a string that is a fully qualified name of a file containing cursor data.

If the high-order word of lpszFileName is zero, the low-order word is a system cursor identifier. The function then searches the [cursors] section in the WIN.INI file for the file associated with the name of that system cursor. For a list of cursor identifiers, see Remarks.

Return Values

If the function is successful, the return value is a handle to the new cursor.

If the function fails, the return value is NULL. To get extended error information, call GetLastError. GetLastError may return the following value:

Value	Meaning
ERROR_FILE_NOT_FOUND	The specified file cannot be found.

Remarks

The following is a list of system cursor names and identifiers.

Cursor Name	Cursor Identifier
"AppStarting"	OCR_APPSTARTING
"Arrow"	OCR_NORMAL
"Crosshair"	OCR_CROSS
"Hand"	Windows 2000: OCR_HAND
"Help"	OCR_HELP
"IBeam"	OCR_IBEAM
"Icon"	OCR_ICON
"No"	OCR_NO
"Size"	OCR_SIZE

"SizeAll"	OCR_SIZEALL
"SizeNESW"	OCR_SIZENESW
"SizeNS"	OCR_SIZENS
"SizeNWSE"	OCR_SIZENWSE
"SizeWE"	OCR_SIZEWE
"UpArrow"	OCR_UP
"Wait"	OCR_WAIT

For example, if the WIN.INI file contains the following

[Cursors]

Arrow = "arrow.ani"

Then the following call causes LoadCursorFromFile to obtain cursor data from the file ARROW.ANI.

LoadCursorFromFile((LPWSTR)OCR_NORMAL)

If the WIN.INI file does not contain an entry for the specified system cursor, the function fails and returns NULL.

Requirements

Windows NT/2000: Requires Windows NT 3.5 or later.

Windows 95/98: Requires Windows 95 or later.

3.289 LoadIcon

The LoadIcon function loads the specified icon resource from the executable (.exe) file associated with an application instance.

Note This function has been superseded by the LoadImage function.

```
LoadIcon: procedure
(
    hInstance       :dword;
    lpIconName      :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__LoadIconA@8" );
```

Parameters

hInstance

[in] Handle to an instance of the module whose executable file contains the icon to be loaded. This parameter must be NULL when a standard icon is being loaded.

lpIconName

[in] Pointer to a null-terminated string that contains the name of the icon resource to be loaded. Alternatively, this parameter can contain the resource identifier in the low-order word and zero in the high-order word. Use the MAKEINTRESOURCE macro to create this value.

To use one of the predefined icons, set the hInstance parameter to NULL and the lpIconName parameter to one of the following values.

Value	Description
IDI_APPLICATION	Default application icon.
IDI_ASTERISK	Same as IDI_INFORMATION.
IDI_ERROR	Hand-shaped icon.
IDI_EXCLAMATION	Same as IDI_WARNING.

IDI_HAND	Same as IDI_ERROR.
IDI_INFORMATION	Asterisk icon.
IDI_QUESTION	Question mark icon.
IDI_WARNING	Exclamation point icon.
IDI_WINLOGO	Windows logo icon.

Return Values

If the function succeeds, the return value is a handle to the newly loaded icon.

If the function fails, the return value is NULL. To get extended error information, call GetLastError.

Remarks

LoadIcon loads the icon resource only if it has not been loaded; otherwise, it retrieves a handle to the existing resource. The function searches the icon resource for the icon most appropriate for the current display. The icon resource can be a color or monochrome bitmap.

LoadIcon can only load an icon whose size conforms to the SM_CXICON and SM_CYICON system metric values. Use the LoadImage function to load icons of other sizes.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.290 LoadImage

The LoadImage function loads an icon, cursor, animated cursor, or bitmap.

```
LoadImage: procedure
(
    hinst           :dword;
    lpszName        :string;
    uType           :dword;
    cxDesired       :dword;
    cyDesired       :dword;
    fuLoad          :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__LoadImageA@24" );
```

Parameters

hinst

[in] Handle to an instance of the module that contains the image to be loaded. To load an OEM image, set this parameter to zero.

lpszName

[in] Specifies the image to load. If the hinst parameter is non-NULL and the fuLoad parameter omits LR_LOADFROMFILE, lpszName specifies the image resource in the hinst module. If the image resource is to be loaded by name, the lpszName parameter is a pointer to a null-terminated string that contains the name of the image resource. If the image resource is to be loaded by ordinal, use the MAKEINTRESOURCE macro to convert the image ordinal into a form that can be passed to the LoadImage function.

If the hinst parameter is NULL and the fuLoad parameter omits the LR_LOADFROMFILE value, the lpszName specifies the OEM image to load. The OEM image identifiers are defined in Winuser.h and have the following prefixes.

Prefix	Meaning
OBM_	OEM bitmaps
OIC_	OEM icons
OCR_	OEM cursors

To pass these constants to the LoadImage function, use the MAKEINTRESOURCE macro. For example, to load the OCR_NORMAL cursor, pass MAKEINTRESOURCE(OCR_NORMAL) as the lpszName parameter and NULL as the hinst parameter.

If the fuLoad parameter includes the LR_LOADFROMFILE value, lpszName is the name of the file that contains the image.

uType

[in] Specifies the type of image to be loaded. This parameter can be one of the following values.

Value	Meaning
IMAGE_BITMAP	Loads a bitmap.
IMAGE_CURSOR	Loads a cursor.
IMAGE_ICON	Loads an icon.

cxDesired

[in] Specifies the width, in pixels, of the icon or cursor. If this parameter is zero and the fuLoad parameter is LR_DEFAULTSIZE, the function uses the SM_CXICON or SM_CXCURSOR system metric value to set the width. If this parameter is zero and LR_DEFAULTSIZE is not used, the function uses the actual resource width.

cyDesired

[in] Specifies the height, in pixels, of the icon or cursor. If this parameter is zero and the fuLoad parameter is LR_DEFAULTSIZE, the function uses the SM_CYICON or SM_CYCURSOR system metric value to set the height. If this parameter is zero and LR_DEFAULTSIZE is not used, the function uses the actual resource height.

fuLoad

[in] This parameter can be one or more of the following values.

Value	Meaning
LR_DEFAULTCOLOR	The default flag; it does nothing. All it means is "not LR_MONOCHROME".
LR_CREATEDIBSECTION	When the uType parameter specifies IMAGE_BITMAP, causes the function to return a DIB section bitmap rather than a compatible bitmap. This flag is useful for loading a bitmap without mapping it to the colors of the display device.

LR_DEFAULTSIZE	Uses the width or height specified by the system metric values for cursors or icons, if the cxDesired or cyDesired values are set to zero. If this flag is not specified and cxDesired and cyDesired are set to zero, the function uses the actual resource size. If the resource contains multiple images, the function uses the size of the first image.
LR_LOADFROMFILE	Loads the image from the file specified by the lpszName parameter. If this flag is not specified, lpszName is the name of the resource.
LR_LOADMAP3DCOLORS	Searches the color table for the image and replaces the following shades of gray with the corresponding 3-D color: Color — Replaced with Dk Gray, RGB(128,128,128) — COLOR_3DSHADOW Gray, RGB(192,192,192) — COLOR_3DFACE Lt Gray, RGB(223,223,223) — COLOR_3DLIGHT Do not use this option if you are loading a bitmap with a color depth greater than 8bpp.

LR_LOADTRANSPARENT	Retrieves the color value of the first pixel in the image and replaces the corresponding entry in the color table with the default window color (COLOR_WINDOW). All pixels in the image that use that entry become the default window color. This value applies only to images that have corresponding color tables. Do not use this option if you are loading a bitmap with a color depth greater than 8bpp. If fuLoad includes both the LR_LOADTRANSPARENT and LR_LOADMAP3DCOLORS values, LRLOADTRANSPARENT takes precedence. However, the color table entry is replaced with COLOR_3DFACE rather than COLOR_WINDOW.
LR_MONOCHROME	Loads the image in black and white.
LR_SHARED	Shares the image handle if the image is loaded multiple times. If LR_SHARED is not set, a second call to for the same resource will load the image again and return a different handle. When you use this flag, the system will destroy the resource when it is no longer needed. Do not use LR_SHARED for images that have non-standard sizes, that may change after loading, or that are loaded from a file. Windows 95/98: The function finds the first image with the requested resource name in the cache, regardless of the size requested.
LR_VGACOLOR	Uses true VGA colors.

Return Values

If the function succeeds, the return value is the handle of the newly loaded image.

If the function fails, the return value is NULL. To get extended error information, call GetLastError.

Remarks

When you are finished using a bitmap, cursor, or icon you loaded without specifying the LR_SHARED flag, you can release its associated memory by calling one of the functions in the following table.

Resource	Release function
Bitmap	DeleteObject
Cursor	**DestroyCursor**
Icon	**DestroyIcon**

The system automatically deletes these resources when the process that created them terminates, however, calling the appropriate function saves memory and decreases the size of the process's working set.

Requirements

Windows NT/2000: Requires Windows NT 4.0 or later.

Windows 95/98: Requires Windows 95 or later.

3.291 LoadKeyboardLayout

The LoadKeyboardLayout function loads a new input locale identifier (formerly called the keyboard layout) into the system. Several input locale identifiers can be loaded at a time, but only one per process is active at a time. Loading multiple input locale identifiers makes it possible to rapidly switch between them.

```
LoadKeyboardLayout: procedure
(
    pwszKLID        :string;
    Flags           :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__LoadKeyboardLayoutA@8" );
```

Parameters

pwszKLID

[in] Pointer to the buffer that specifies the name of the input locale identifier to load. This name is a string composed of the hexadecimal value of the language identifier (low word) and a device identifier (high word). For example, U.S. English has a language identifier of 0x0409, so the primary U.S. English layout is named "00000409". Variants of U.S. English layout (such as the Dvorak layout) are named "00010409", "00020409", and so on.

Flags

[in] Specifies how the input locale identifier is to be loaded. This parameter can be one of the following values.

Value	Meaning
KLF_ACTIVATE	If the specified input locale identifier is not already loaded, the function loads and activates the input locale identifier for the current thread.

KLF_NOTELLSHELL	Prevents a ShellProc hook procedure from receiving an HSHELL_LANGUAGE hook code when the new input locale identifier is loaded. This value is typically used when an application loads multiple input locale identifiers one after another. Applying this value to all but the last input locale identifier delays the shell's processing until all input locale identifiers have been added.
KLF_REORDER	Moves the specified input locale identifier to the head of the input locale identifier list, making that locale identifier the active locale identifier for the current thread. This value reorders the input locale identifier list even if KLF_ACTIVATE is not provided.
KLF_REPLACELANG	Windows 95/98, Windows NT 4.0, and Windows 2000: If the new input locale identifier has the same language identifier as a current input locale identifier, the new input locale identifier replaces the current one as the input locale identifier for that language. If this value is not provided and the input locale identifiers have the same language identifiers, the current input locale identifier is not replaced and the function returns NULL.
KLF_SUBSTITUTE_OK	Substitutes the specified input locale identifier with another locale preferred by the user. The system starts with this flag set, and it is recommended that your application always use this flag. The substitution occurs only if the registry key HKEY_CURRENT_USER\Keyboard Layout\Substitutes explicitly defines a substitution locale. For example, if the key includes the value name "00000409" with value "00010409", loading the U.S. English layout ("00000409") causes the Dvorak U.S. English layout ("00010409") to be loaded instead. The system uses KLF_SUBSTITUTE_OK when booting, and it is recommended that all applications use this value when loading input locale identifiers to ensure that the user's preference is selected.
KLF_SETFORPROCESS	Windows 2000:: This flag is valid only with KLF_ACTIVATE. Activates the specified input locale identifier for the entire process and sends the WM_INPUTLANGCHANGE message to the current thread's Focus or Active window. Typically, activates an input locale identifier only for the current thread.
KLF_UNLOADPREVIOUS	This flag is unsupported. Use the UnloadKeyboardLayout function instead.

Return Values

If the function succeeds, the return value is the input locale identifier to the locale matched with the requested name. If no matching locale is available, the return value is NULL. To get extended error information, call GetLastError.

Remarks

The input locale identifier is a broader concept than a keyboard layout, since it can also encompass a speech-to-text converter, an IME, or any other form of input.

An application can and will typically load the default input locale identifier or IME for a language and can do so by specifying only a string version of the language identifier. If an application wants to load a specific locale or IME, it should read the registry to determine the specific input locale identifier to pass to LoadKeyboardLayout. In this case, a request to activate the default input locale identifier for a locale will activate the first matching one.

A specific IME should be activated using an explicit input locale identifier returned from GetKeyboardLayout, GetKeyboardLayoutList, or LoadKeyboardLayout.

Windows 95/98: If an input locale identifier is to be loaded with the same language as a previously loaded input locale identifier and the KLF_REPLACELANG flag is not set, the call fails. Only one loaded locale may be associated with a language. (It is acceptable for multiple IMEs to be loaded with associations to the same language.)

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.292 LoadMenu

The LoadMenu function loads the specified menu resource from the executable (.exe) file associated with an application instance.

```
LoadMenu: procedure
(
    hInstance       :dword;
    lpMenuName      :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__LoadMenuA@8" );
```

Parameters

hInstance

[in] Handle to the module containing the menu resource to be loaded.

lpMenuName

[in] Pointer to a null-terminated string that contains the name of the menu resource. Alternatively, this parameter can consist of the resource identifier in the low-order word and zero in the high-order word. To create this value, use the MAKEINTRESOURCE macro.

Return Values

If the function succeeds, the return value is a handle to the menu resource.

If the function fails, the return value is NULL. To get extended error information, call GetLastError.

Remarks

The DestroyMenu function is used, before an application closes, to destroy the menu and free memory that the loaded menu occupied.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.293 LoadMenuIndirect

The LoadMenuIndirect function loads the specified menu template in memory.

```
LoadMenuIndirect: procedure
```

```
(
    var lpMenuTemplate  :MENUTEMPLATE
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__LoadMenuIndirectA@4" );
```

Parameters

lpMenuTemplate

[in] Pointer to a menu template or an extended menu template.

A menu template consists of a MENUITEMTEMPLATEHEADER structure followed by one or more contiguous MENUITEMTEMPLATE structures. An extended menu template consists of a MENUEX_TEMPLATE_HEADER structure followed by one or more contiguous MENUEX_TEMPLATE_ITEM structures.

Return Values

If the function succeeds, the return value is a handle to the menu.

If the function fails, the return value is NULL. To get extended error information, call GetLastError.

Remarks

For both the ANSI and the Unicode version of this function, the strings in the MENUITEMTEMPLATE structure must be Unicode strings.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.294 LoadString

The LoadString function loads a string resource from the executable file associated with a specified module, copies the string into a buffer, and appends a terminating null character.

```
LoadString: procedure
(
        hInstance   :dword;
        uID         :dword;
    var lpBuffer    :var;
        nBuffermax  :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__LoadStringA@16" );
```

Parameters

hInstance

[in] Handle to an instance of the module whose executable file contains the string resource.

uID

[in] Specifies the integer identifier of the string to be loaded.

lpBuffer

[out] Pointer to the buffer to receive the string.

nBufferMax

[in] Specifies the size of the buffer, in TCHARs. This refers to bytes for ANSI versions of the function or characters for Unicode versions. The string is truncated and null terminated if it is longer than the number of characters specified.

Return Values

If the function succeeds, the return value is the number of TCHARs copied into the buffer, not including the null-terminating character, or zero if the string resource does not exist. To get extended error information, call GetLastError.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.295 LockWindowUpdate

The LockWindowUpdate function disables or reenables drawing in the specified window. Only one window can be locked at a time.

```
LockWindowUpdate: procedure
(
    hWndLock        :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__LockWindowUpdate@4" );
```

Parameters

hWndLock

[in] Specifies the window in which drawing will be disabled. If this parameter is NULL, drawing in the locked window is enabled.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero, indicating that an error occurred or another window was already locked.

Windows NT/ 2000: To get extended error information, call GetLastError.

Remarks

If an application with a locked window (or any locked child windows) calls the GetDC, GetDCEx, or BeginPaint function, the called function returns a device context with a visible region that is empty. This will occur until the application unlocks the window by calling LockWindowUpdate, specifying a value of NULL for hWndLock.

If an application attempts to draw within a locked window, the system records the extent of the attempted operation in a bounding rectangle. When the window is unlocked, the system invalidates the area within this bounding rectangle, forcing an eventual WM_PAINT message to be sent to the previously locked window and its child windows. If no drawing has occurred while the window updates were locked, no area is invalidated.

LockWindowUpdate does not make the specified window invisible and does not clear the WS_VISIBLE style bit.

A locked window cannot be moved.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.296 LockWorkStation

The LockWorkStation function locks the workstation's display, protecting it from unauthorized use.

```
LockWorkStation: procedure;
    @stdcall;
    @returns( "eax" );
    @external( "__imp__LockWorkStation@0" );
```

Parameters

This function has no parameters.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

This function has the same result as pressing Ctrl+Alt+Del and clicking Lock Workstation. To unlock the workstation, the user must log in.

Common reasons for failure include no user is logged on, the workstation is already locked, the process is not running on the interactive desktop, or the request is denied by the Graphical Identification and Authentication (GINA) DLL.

Requirements

Windows NT/2000: Requires Windows 2000 or later.

Windows 95/98: Unsupported.

3.297 LookupIconIdFromDirectory

The LookupIconIdFromDirectory function searches through icon or cursor data for the icon or cursor that best fits the current display device.

To specify a desired height or width, use the LookupIconIdFromDirectoryEx function.

```
LookupIconIdFromDirectory: procedure
(
    var presbits    :var;
        fIcon       :boolean
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__LookupIconIdFromDirectory@8" );
```

Parameters

presbits

[in] Pointer to the icon or cursor directory data. Because this function does not validate the resource data, it causes a general protection (GP) fault or returns an undefined value if presbits is not pointing to valid resource data.

fIcon

[in] Specifies whether an icon or a cursor is sought. If this parameter is TRUE, the function is searching for an icon; if the parameter is FALSE, the function is searching for a cursor.

Return Values

If the function succeeds, the return value is an integer resource identifier for the icon or cursor that best fits the current display device.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

A resource file of type RT_GROUP_ICON (RT_GROUP_CURSOR indicates cursors) contains icon (or cursor) data in several device-dependent and device-independent formats. LookupIconIdFromDirectory searches the resource file for the icon (or cursor) that best fits the current display device and returns its integer identifier. The FindResource and FindResourceEx functions use the MAKEINTRESOURCE macro with this identifier to locate the resource in the module.

The icon directory is loaded from a resource file with resource type RT_GROUP_ICON (or RT_GROUP_CURSOR for cursors), and an integer resource name for the specific icon to be loaded. LookupIconIdFromDirectory returns an integer identifier that is the resource name of the icon that best fits the current display device.

The LoadIcon, LoadCursor, and LoadImage functions use this function to search the specified resource data for the icon or cursor that best fits the current display device.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.298 LookupIconIdFromDirectoryEx

The LookupIconIdFromDirectoryEx function searches through icon or cursor data for the icon or cursor that best fits the current display device.

```
LookupIconIdFromDirectoryEx: procedure
(
    var presbits    :var;
        fIcon       :boolean;
        cxDesired   :dword;
        cyDesired   :dword;
        Flags       :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__LookupIconIdFromDirectoryEx@20" );
```

Parameters

presbits

[in] Pointer to the icon or cursor directory data. Because this function does not validate the resource data, it causes a general protection (GP) fault or returns an undefined value if presbits is not pointing to valid resource data.

fIcon

[in] Specifies whether an icon or a cursor is sought. If this parameter is TRUE, the function is searching for an icon; if the parameter is FALSE, the function is searching for a cursor.

cxDesired

[in] Specifies the desired width, in pixels, of the icon. If this parameter is zero, the function uses the SM_CXICON or SM_CXCURSOR system metric value.

cyDesired

[in] Specifies the desired height, in pixels, of the icon. If this parameter is zero, the function uses the SM_CYICON or SM_CYCURSOR system metric value.

Flags

[in] Specifies a combination of the following values:

Value	Meaning
LR_DEFAULTCOLOR	Uses the default color format.
LR_MONOCHROME	Creates a monochrome icon or cursor.

Return Values

If the function succeeds, the return value is an integer resource identifier for the icon or cursor that best fits the current display device.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

A resource file of type RT_GROUP_ICON (RT_GROUP_CURSOR indicates cursors) contains icon (or cursor) data in several device-dependent and device-independent formats. LookupIconIdFromDirectoryEx searches the resource file for the icon (or cursor) that best fits the current display device and returns its integer identifier. The FindResource and FindResourceEx functions use the MAKEINTRESOURCE macro with this identifier to locate the resource in the module.

The icon directory is loaded from a resource file with resource type RT_GROUP_ICON (or RT_GROUP_CURSOR for cursors), and an integer resource name for the specific icon to be loaded. LookupIconIdFromDirectoryEx returns an integer identifier that is the resource name of the icon that best fits the current display device.

The LoadIcon, LoadImage, and LoadCursor functions use this function to search the specified resource data for the icon or cursor that best fits the current display device.

Requirements

Windows NT/2000: Requires Windows NT 4.0 or later.

Windows 95/98: Requires Windows 95 or later.

3.299 MapDialogRect

The MapDialogRect function converts the specified dialog box units to screen units (pixels). The function replaces the coordinates in the specified RECT structure with the converted coordinates, which allows the structure to be used to create a dialog box or position a control within a dialog box.

```
MapDialogRect: procedure
```

```
(
        hDlg        :dword;
    var lpRect      :RECT
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__MapDialogRect@8" );
```

Parameters

hDlg

[in] Handle to a dialog box. This function accepts only handles returned by one of the dialog box creation functions; handles for other windows are not valid.

lpRect

[in/out] Pointer to a RECT structure that contains the dialog box coordinates to be converted.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

The MapDialogRect function assumes that the initial coordinates in the RECT structure represent dialog box units. To convert these coordinates from dialog box units to pixels, the function retrieves the current horizontal and vertical base units for the dialog box, then applies the following formulas:

left = MulDiv(left, baseunitX, 4);

right = MulDiv(right, baseunitX, 4);

top = MulDiv(top, baseunitY, 8);

bottom = MulDiv(bottom, baseunitY, 8);

If the dialog box template has the DS_SETFONT or DS_SHELLFONT style, the base units are the average width and height, in pixels, of the characters in the font specified by the template.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.300 MapVirtualKey

The MapVirtualKey function translates (maps) a virtual-key code into a scan code or character value, or translates a scan code into a virtual-key code.

To specify a handle to the keyboard layout to use for translating the specified code, use the MapVirtualKeyEx function.

```
MapVirtualKey: procedure
(
    uCode           :dword;
    uMapType        :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__MapVirtualKeyA@8" );
```

Parameters

uCode

[in] Specifies the virtual-key code or scan code for a key. How this value is interpreted depends on the value of the uMapType parameter.

uMapType

[in] Specifies the translation to perform. The value of this parameter depends on the value of the uCode parameter.

Value	Meaning
0	uCode is a virtual-key code and is translated into a scan code. If it is a virtual-key code that does not distinguish between left- and right-hand keys, the left-hand scan code is returned. If there is no translation, the function returns 0.
1	uCode is a scan code and is translated into a virtual-key code that does not distinguish between left- and right-hand keys. If there is no translation, the function returns 0.
2	uCode is a virtual-key code and is translated into an unshifted character value in the low-order word of the return value. Dead keys (diacritics) are indicated by setting the top bit of the return value. If there is no translation, the function returns 0.
3	Windows NT/2000: uCode is a scan code and is translated into a virtual-key code that distinguishes between left- and right-hand keys. If there is no translation, the function returns 0.

Return Values

The return value is either a scan code, a virtual-key code, or a character value, depending on the value of uCode and uMapType. If there is no translation, the return value is zero.

Remarks

An application can use MapVirtualKey to translate scan codes to the virtual-key code constants VK_SHIFT, VK_CONTROL, and VK_MENU, and vice versa. These translations do not distinguish between the left and right instances of the SHIFT, CTRL, or ALT keys.

Windows NT/2000: An application can get the scan code corresponding to the left or right instance of one of these keys by calling MapVirtualKey with uCode set to one of the following virtual-key code constants.

VK_LSHIFT

VK_RSHIFT

VK_LCONTROL

VK_RCONTROL

VK_LMENU

VK_RMENU

These left- and right-distinguishing constants are available to an application only through the GetKeyboardState, SetKeyboardState, GetAsyncKeyState, GetKeyState, and MapVirtualKey functions.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.301 MapVirtualKeyEx

The MapVirtualKeyEx function translates (maps) a virtual-key code into a scan code or character value, or translates a scan code into a virtual-key code. The function translates the codes using the input language and an input locale identifier.

```
MapVirtualKeyEx: procedure
(
    uCode           :dword;
    uMapType        :dword;
    dwhkl           :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__MapVirtualKeyExA@12" );
```

Parameters

uCode

[in] Specifies the virtual-key code or scan code for a key. How this value is interpreted depends on the value of the uMapType parameter.

uMapType

[in] Specifies the translation to perform. The value of this parameter depends on the value of the uCode parameter.

Value	Meaning
0	uCode is a virtual-key code and is translated into a scan code. If it is a virtual-key code that does not distinguish between left- and right-hand keys, the left-hand scan code is returned. If there is no translation, the function returns 0.
1	uCode is a scan code and is translated into a virtual-key code that does not distinguish between left- and right-hand keys. If there is no translation, the function returns 0.
2	uCode is a virtual-key code and is translated into an unshifted character value in the low order word of the return value. Dead keys (diacritics) are indicated by setting the top bit of the return value. If there is no translation, the function returns 0.
3	Windows NT/2000: uCode is a scan code and is translated into a virtual-key code that distinguishes between left- and right-hand keys. If there is no translation, the function returns 0.

dwhkl

[in] Input locale identifier to use for translating the specified code. This parameter can be any input locale identifier previously returned by the LoadKeyboardLayout function.

Return Values

The return value is either a scan code, a virtual-key code, or a character value, depending on the value of uCode and uMapType. If there is no translation, the return value is zero.

Remarks

The input locale identifier is a broader concept than a keyboard layout, since it can also encompass a speech-to-text converter, an IME, or any other form of input.

An application can use MapVirtualKeyEx to translate scan codes to the virtual-key code constants VK_SHIFT, VK_CONTROL, and VK_MENU, and vice versa. These translations do not distinguish between the left and

right instances of the SHIFT, CTRL, or ALT keys.

Windows NT/2000: An application can get the scan code corresponding to the left or right instance of one of these keys by calling MapVirtualKeyEx with uCode set to one of the following virtual-key code constants.

VK_LSHIFT

VK_RSHIFT

VK_LCONTROL

VK_RCONTROL

VK_LMENU

VK_RMENU

These left- and right-distinguishing constants are available to an application only through the GetKeyboardState, SetKeyboardState, GetAsyncKeyState, GetKeyState, MapVirtualKey, and MapVirtualKeyEx functions. For list complete table of virtual key codes, see Virtual-Key Codes .

Requirements

Windows NT/2000: Requires Windows NT 4.0 or later.

Windows 95/98: Requires Windows 95 or later.

3.302 MapWindowPoints

The MapWindowPoints function converts (maps) a set of points from a coordinate space relative to one window to a coordinate space relative to another window.

```
MapWindowPoints: procedure
(
        hWndFrom        :dword;
        hWndTo          :dword;
    var lpPoints        :var;
        cPoints         :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__MapWindowPoints@16" );
```

Parameters

hWndFrom

[in] Handle to the window from which points are converted. If this parameter is NULL or HWND_DESKTOP, the points are presumed to be in screen coordinates.

hWndTo

[in] Handle to the window to which points are converted. If this parameter is NULL or HWND_DESKTOP, the points are converted to screen coordinates.

lpPoints

[in/out] Pointer to an array of POINT structures that contain the set of points to be converted. The points are in device units. This parameter can also point to a RECT structure, in which case the cPoints parameter should be set to 2.

cPoints

[in] Specifies the number of POINT structures in the array pointed to by the lpPoints parameter.

Return Values

If the function succeeds, the low-order word of the return value is the number of pixels added to the horizontal coordinate of each source point in order to compute the horizontal coordinate of each destination point; the high-order word is the number of pixels added to the vertical coordinate of each source point in order to compute the vertical coordinate of each destination point.

If the function fails, the return value is zero.

Windows NT/ 2000: To get extended error information, call GetLastError.

Remarks

If hWndFrom or hWndTo (or both) are mirrored windows (that is, have WS_EX_LAYOUTRTL extended style), MapWindowPoints will automatically adjust mirrored coordinates if you pass two or less points in lpPoints. If you pass more than two points, the function will not fail but it will return erroneous positions. Thus, to guarantee the correct transformation of rectangle coordinates, you must call MapWindowPoints with two or less points at a time, as shown in the following example:

```
    RECT        rc[10];

   for(int i =0; i < (sizeof(rc)/sizeof(rc[0])); i++)
   {
       MapWindowPoints(hWnd1, hWnd2, (LPPOINT)(&rc[i]), (sizeof(RECT)/sizeof(POINT)) );
   }
```

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.303 MenuItemFromPoint

The MenuItemFromPoint function determines which menu item, if any, is at the specified location.

```
MenuItemFromPoint: procedure
(
    hWnd            :dword;
    hMenu           :dword;
    ptScreen        :POINT
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__MenuItemFromPoint@16" );
```

Parameters

hWnd

[in] Handle to the window containing the menu.

Windows 98 and Windows 2000: If this value is NULL and the hMenu parameter represents a popup menu, the function will find the menu window.

hMenu

[in] Handle to the menu containing the menu items to hit test.

ptScreen

[in] A POINT structure that specifies the location to test. If hMenu specifies a menu bar, this parameter is in window coordinates. Otherwise, it is in client coordinates.

Return Values

Returns the zero-based position of the menu item at the specified location or –1 if no menu item is at the specified location.

Requirements

Windows NT/2000: Requires Windows NT 4.0 or later.

Windows 95/98: Requires Windows 95 or later.

3.304 MessageBeep

The MessageBeep function plays a waveform sound. The waveform sound for each sound type is identified by an entry in the registry.

```
MessageBeep: procedure
(
    uType           :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__MessageBeep@4" );
```

Parameters

uType

[in] Specifies the sound type, as identified by an entry in the registry. This parameter can be one of the following values.

Value	Sound
-1	Standard beep using the computer speaker
MB_ICONASTERISK	SystemAsterisk
MB_ICONEXCLAMATION	SystemExclamation
MB_ICONHAND	SystemHand
MB_ICONQUESTION	SystemQuestion
MB_OK	SystemDefault

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

After queuing the sound, the MessageBeep function returns control to the calling function and plays the sound asynchronously.

If it cannot play the specified alert sound, MessageBeep attempts to play the system default sound. If it cannot play the system default sound, the function produces a standard beep sound through the computer speaker.

The user can disable the warning beep by using the Sound control panel application.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.305 MessageBox

The MessageBox function creates, displays, and operates a message box. The message box contains an application-defined message and title, plus any combination of predefined icons and push buttons.

```
MessageBox: procedure
(
    hWnd            :dword;
    lpText          :string;
    lpCaption       :string;
    uType           :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__MessageBoxA@16" );
```

Parameters

hWnd

[in] Handle to the owner window of the message box to be created. If this parameter is NULL, the message box has no owner window.

lpText

[in] Pointer to a null-terminated string that contains the message to be displayed.

lpCaption

[in] Pointer to a null-terminated string that contains the dialog box title. If this parameter is NULL, the default title Error is used.

uType

[in] Specifies the contents and behavior of the dialog box. This parameter can be a combination of flags from the following groups of flags.

To indicate the buttons displayed in the message box, specify one of the following values.

Value	Meaning
MB_ABORTRETRYIGNORE	The message box contains three push buttons: Abort, Retry, and Ignore.
MB_CANCELTRYCONTINUE	Windows 2000: The message box contains three push buttons: Cancel, Try Again, Continue. Use this message box type instead of MB_ABORTRETRYIGNORE.
MB_HELP	Windows 95/98, Windows NT 4.0 and later: Adds a Help button to the message box. When the user clicks the Help button or presses F1, the system sends a WM_HELP message to the owner.
MB_OK	The message box contains one push button: OK. This is the default.
MB_OKCANCEL	The message box contains two push buttons: OK and Cancel.

MB_RETRYCANCEL	The message box contains two push buttons: Retry and Cancel.
MB_YESNO	The message box contains two push buttons: Yes and No.
MB_YESNOCANCEL	The message box contains three push buttons: Yes, No, and Cancel.

To display an icon in the message box, specify one of the following values.

Value	Meaning
MB_ICONEXCLAMATION,	An exclamation-point icon appears in the message box.
MB_ICONWARNING	
MB_ICONINFORMATION,	An icon consisting of a lowercase letter i in a circle appears in the message box.
MB_ICONASTERISK	
MB_ICONQUESTION	A question-mark icon appears in the message box.
MB_ICONSTOP,	A stop-sign icon appears in the message box.
MB_ICONERROR,	
MB_ICONHAND	

To indicate the default button, specify one of the following values.

Value	Meaning
MB_DEFBUTTON1	The first button is the default button. MB_DEFBUTTON1 is the default unless MB_DEFBUTTON2, MB_DEFBUTTON3, or MB_DEFBUTTON4 is specified.
MB_DEFBUTTON2	The second button is the default button.
MB_DEFBUTTON3	The third button is the default button.
MB_DEFBUTTON4	The fourth button is the default button.

To indicate the modality of the dialog box, specify one of the following values.

Value	Meaning
MB_APPLMODAL	The user must respond to the message box before continuing work in the window identified by the hWnd parameter. However, the user can move to the windows of other threads and work in those windows. Depending on the hierarchy of windows in the application, the user may be able to move to other windows within the thread. All child windows of the parent of the message box are automatically disabled, but popup windows are not. MB_APPLMODAL is the default if neither MB_SYSTEMMODAL nor MB_TASKMODAL is specified.
MB_SYSTEMMODAL	Same as MB_APPLMODAL except that the message box has the WS_EX_TOPMOST style. Use system-modal message boxes to notify the user of serious, potentially damaging errors that require immediate attention (for example, running out of memory). This flag has no effect on the user's ability to interact with windows other than those associated with hWnd.

MB_TASKMODAL	Same as MB_APPLMODAL except that all the top-level windows belonging to the current thread are disabled if the hWnd parameter is NULL. Use this flag when the calling application or library does not have a window handle available but still needs to prevent input to other windows in the calling thread without suspending other threads.

To specify other options, use one or more of the following values.

Value	Meaning
MB_DEFAULT_DESKTOP_ONLY	Windows NT/2000: Same as MB_SERVICE_NOTIFICATION except that the system will display the message box only on the default desktop of the interactive window station. For more information, see Window Stations and Desktops. Windows NT 4.0 and earlier: If the current input desktop is not the default desktop, fails. Windows 2000: If the current input desktop is not the default desktop, does not return until the user switches to the default desktop. Windows 95/98: This flag has no effect.
MB_RIGHT	The text is right-justified.
MB_RTLREADING	Displays message and caption text using right-to-left reading order on Hebrew and Arabic systems.
MB_SETFOREGROUND	The message box becomes the foreground window. Internally, the system calls the SetForegroundWindow function for the message box.
MB_TOPMOST	The message box is created with the WS_EX_TOPMOST window style.
MB_SERVICE_NOTIFICATION	Windows NT/2000: The caller is a service notifying the user of an event. The function displays a message box on the current active desktop, even if there is no user logged on to the computer. Terminal Services: If the calling thread has an impersonation token, the function directs the message box to the session specified in the impersonation token. If this flag is set, the hWnd parameter must be NULL. This is so the message box can appear on a desktop other than the desktop corresponding to the hWnd. For more information on the changes between Windows NT 3.51 and Windows NT 4.0, see Remarks.
MB_SERVICE_NOTIFICATION_NT3X	Windows NT/2000: This value corresponds to the value defined for MB_SERVICE_NOTIFICATION for Windows NT version 3.51. For more information on the changes between Windows NT 3.51 and Windows NT 4.0, see Remarks.

Return Values

If the function succeeds, the return value is one of the following menu-item values.

Value	Meaning
IDABORT	Abort button was selected.
IDCANCEL	Cancel button was selected.
IDCONTINUE	Continue button was selected.
IDIGNORE	Ignore button was selected.

IDNO	No button was selected.
IDOK	OK button was selected.
IDRETRY	Retry button was selected.
IDTRYAGAIN	Try Again button was selected.
IDYES	Yes button was selected.

If a message box has a Cancel button, the function returns the IDCANCEL value if either the ESC key is pressed or the Cancel button is selected. If the message box has no Cancel button, pressing ESC has no effect.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

When you use a system-modal message box to indicate that the system is low on memory, the strings pointed to by the lpText and lpCaption parameters should not be taken from a resource file, because an attempt to load the resource may fail.

If you create a message box while a dialog box is present, use a handle to the dialog box as the hWnd parameter. The hWnd parameter should not identify a child window, such as a control in a dialog box.

Windows 95: The system can support a maximum of 16,364 window handles.

Windows NT/ 2000: The value of MB_SERVICE_NOTIFICATION changed starting with Windows NT 4.0. Windows NT 4.0 provides backward compatibility for pre-existing services by mapping the old value to the new value in the implementation of MessageBox. This mapping is only done for executables that have a version number less than 4.0, as set by the linker.

To build a service that uses MB_SERVICE_NOTIFICATION, and can run on both Windows NT 3.x and Windows NT 4.0, you can do one of the following.

- At link-time, specify a version number less than 4.0
- At link-time, specify version 4.0. At run-time, use the GetVersionEx function to check the system version. Then when running on Windows NT 3.x, use MB_SERVICE_NOTIFICATION_NT3X; and on Windows NT 4.0, use MB_SERVICE_NOTIFICATION.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.306 MessageBoxEx

The MessageBoxEx function creates, displays, and operates a message box. The message box contains an application-defined message and title, plus any combination of predefined icons and push buttons. The buttons are in the language of the system user interface.

Currently MessageBoxEx and MessageBox work the same way.

```
MessageBoxEx: procedure
(
    hWnd            :dword;
    lpText          :string;
    lpCaption       :string;
    yType           :dword;
    wLanguageID     :word
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__MessageBoxExA@20" );
```

Parameters

hWnd

[in] Handle to the owner window of the message box to be created. If this parameter is NULL, the message box has no owner window.

lpText

[in] Pointer to a null-terminated string that contains the message to be displayed.

lpCaption

[in] Pointer to a null-terminated string that contains the dialog box title. If this parameter is NULL, the default title Error is used.

uType

[in] Specifies the contents and behavior of the dialog box. This parameter can be a combination of flags from the following groups of flags.

To indicate the buttons displayed in the message box, specify one of the following values.

Value	Meaning
MB_ABORTRETRYIGNORE	The message box contains three push buttons: Abort, Retry, and Ignore.
MB_CANCELTRYCONTINUE	Windows 2000: The message box contains three push buttons: Cancel, Try Again, Continue. Use this message box type instead of MB_ABORTRETRYIGNORE.
MB_HELP	Windows 95/98, Windows NT 4.0 and later: Adds a Help button to the message box. When the user clicks the Help button or presses F1, the system sends a WM_HELP message to the owner.
MB_OK	The message box contains one push button: OK. This is the default.
MB_OKCANCEL	The message box contains two push buttons: OK and Cancel.
MB_RETRYCANCEL	The message box contains two push buttons: Retry and Cancel.
MB_YESNO	The message box contains two push buttons: Yes and No.
MB_YESNOCANCEL	The message box contains three push buttons: Yes, No, and Cancel.

To display an icon in the message box, specify one of the following values.

Value	Meaning
MB_ICONEXCLAMATION, MB_ICONWARNING	An exclamation-point icon appears in the message box.
MB_ICONINFORMATION, MB_ICONASTERISK	An icon consisting of a lowercase letter i in a circle appears in the message box.
MB_ICONQUESTION	A question-mark icon appears in the message box.
MB_ICONSTOP, MB_ICONERROR, MB_ICONHAND	A stop-sign icon appears in the message box.

To indicate the default button, specify one of the following values.

Value	Meaning

MB_DEFBUTTON1	The first button is the default button. MB_DEFBUTTON1 is the default unless MB_DEFBUTTON2, MB_DEFBUTTON3, or MB_DEFBUTTON4 is specified.
MB_DEFBUTTON2	The second button is the default button.
MB_DEFBUTTON3	The third button is the default button.
MB_DEFBUTTON4	The fourth button is the default button.

To indicate the modality of the dialog box, specify one of the following values.

Value	Meaning
MB_APPLMODAL	The user must respond to the message box before continuing work in the window identified by the hWnd parameter. However, the user can move to the windows of other threads and work in those windows. Depending on the hierarchy of windows in the application, the user may be able to move to other windows within the thread. All child windows of the parent of the message box are automatically disabled, but popup windows are not. MB_APPLMODAL is the default if neither MB_SYSTEMMODAL nor MB_TASKMODAL is specified.
MB_SYSTEMMODAL	Same as MB_APPLMODAL except that the message box has the WS_EX_TOPMOST style. Use system-modal message boxes to notify the user of serious, potentially damaging errors that require immediate attention (for example, running out of memory). This flag has no effect on the user's ability to interact with windows other than those associated with hWnd.
MB_TASKMODAL	Same as MB_APPLMODAL except that all the top-level windows belonging to the current thread are disabled if the hWnd parameter is NULL. Use this flag when the calling application or library does not have a window handle available but still needs to prevent input to other windows in the calling thread without suspending other threads.

To specify other options, use one or more of the following values.

Values	Meaning
MB_DEFAULT_DESKTOP_ONLY	Windows NT/ 2000: Same as MB_SERVICE_NOTIFICATION except that the system will display the message box only on the default desktop of the interactive window station. For more information, see Window Stations and Desktops. Windows NT 4.0 and earlier: If the current input desktop is not the default desktop, fails. Windows 2000: If the current input desktop is not the default desktop, does not return until the user switches to the default desktop. Windows 95/98: This flag has no effect.
MB_RIGHT	The text is right-justified.
MB_RTLREADING	Displays message and caption text using right-to-left reading order on Hebrew and Arabic systems.

MB_SETFOREGROUND	The message box becomes the foreground window. Internally, the system calls the SetForegroundWindow function for the message box.
MB_TOPMOST	The message box is created with the WS_EX_TOPMOST window style.
MB_SERVICE_NOTIFICATION	Windows NT/ 2000: The caller is a service notifying the user of an event. The function displays a message box on the current active desktop, even if there is no user logged on to the computer. Terminal Services: If the calling thread has an impersonation token, the function directs the message box to the session specified in the impersonation token. If this flag is set, the hWnd parameter must be NULL. This is so the message box can appear on a desktop other than the desktop corresponding to the hWnd. For more information on the changes between Windows NT 3.51 and Windows NT 4.0, see Remarks.
MB_SERVICE_NOTIFICATION_NT3X	Windows NT/2000: This value corresponds to the value defined for MB_SERVICE_NOTIFICATION for Windows NT version 3.51. For more information on the changes between Windows NT 3.51 and Windows NT 4.0, see Remarks.

wLanguageId
[in] Reserved.

Return Values

If the function succeeds, the return value is one of the following menu-item values.

Value	Meaning
IDABORT	Abort button was selected.
IDCANCEL	Cancel button was selected.
IDCONTINUE	Continue button was selected.
IDIGNORE	Ignore button was selected.
IDNO	No button was selected.
IDOK	OK button was selected.
IDRETRY	Retry button was selected.
IDTRYAGAIN	Try Again button was selected.
IDYES	Yes button was selected.

If a message box has a Cancel button, the function returns the IDCANCEL value when either the ESC key or Cancel button is pressed. If the message box has no Cancel button, pressing the ESC key has no effect.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

When you create a system-modal message box to indicate that the system is low on memory, the strings passed as the lpText and lpCaption parameters should not be taken from a resource file, because an attempt to load the resource may fail.

If you create a message box while a dialog box is present, use a handle to the dialog box as the hWnd parameter. The hWnd parameter should not identify a child window, such as a dialog box.

Windows 95: The system can support a maximum of 16,364 window handles.

Windows NT/ 2000: The value of MB_SERVICE_NOTIFICATION changed starting with Windows NT 4.0. Windows NT 4.0 provides backward compatibility for pre-existing services by mapping the old value to the new value in the implementation of MessageBoxEx. This mapping is only done for executables that have a version

number less than 4.0, as set by the linker.

To build a service that uses MB_SERVICE_NOTIFICATION, and can run on both Windows NT 3.x and Windows NT 4.0, you can do one of the following.

- At link-time, specify a version number less than 4.0
- At link-time, specify version 4.0. At run-time, use the GetVersionEx function to check the system version. Then when running on Windows NT 3.x, use MB_SERVICE_NOTIFICATION_NT3X; and on Windows NT 4.0, use MB_SERVICE_NOTIFICATION.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.307 MessageBoxIndirect

The MessageBoxIndirect function creates, displays, and operates a message box. The message box contains application-defined message text and title, any icon, and any combination of predefined push buttons.

```
MessageBoxIndirect: procedure
(
    var lpMsgBoxParams  :MSGBOXPARAMS
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__MessageBoxIndirectA@4" );
```

Parameters

lpMsgBoxParams

[in] Pointer to a MSGBOXPARAMS structure that contains information used to display the message box.

Return Values

If the function succeeds, the return value is one of the following menu-item values.

Value	Meaning
IDABORT	Abort button was selected.
IDCANCEL	Cancel button was selected.
IDCONTINUE	Continue button was selected.
IDIGNORE	Ignore button was selected.
IDNO	No button was selected.
IDOK	OK button was selected.
IDRETRY	Retry button was selected.
IDTRYAGAIN	Try Again button was selected.
IDYES	Yes button was selected.

If a message box has a Cancel button, the function returns the IDCANCEL value if either the ESC key is pressed or the Cancel button is selected. If the message box has no Cancel button, pressing ESC has no effect.

If there is not enough memory to create the message box, the return value is zero.

Remarks

When you use a system-modal message box to indicate that the system is low on memory, the strings pointed to by the lpszText and lpszCaption members of the MSGBOXPARAMS structure should not be taken from a resource file, because an attempt to load the resource may fail.

If you create a message box while a dialog box is present, use a handle to the dialog box as the hWnd parameter. The hWnd parameter should not identify a child window, such as a control in a dialog box.

Windows 95: The system can support a maximum of 16,364 window handles.

Requirements

Windows NT/2000: Requires Windows NT 4.0 or later.

Windows 95/98: Requires Windows 95 or later.

3.308 ModifyMenu

The ModifyMenu function changes an existing menu item. This function is used to specify the content, appearance, and behavior of the menu item.

Note The ModifyMenu function has been superseded by the SetMenuItemInfo function. You can still use ModifyMenu, however, if you do not need any of the extended features of SetMenuItemInfo.

```
ModifyMenu: procedure
(
    hMnu            :dword;
    uPosition       :dword;
    uFlags          :dword;
    uIDNewItem      :dword;
    lpNewItem       :string
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__ModifyMenuA@20" );
```

Parameters

hMnu

[in] Handle to the menu to be changed.

uPosition

[in] Specifies the menu item to be changed, as determined by the uFlags parameter.

uFlags

[in] Specifies flags that control the interpretation of the uPosition parameter and the content, appearance, and behavior of the menu item. This parameter must be a combination of one of the following required values and at least one of the values listed in the following Remarks section.

Value	Meaning
MF_BYCOMMAND	Indicates that the uPosition parameter gives the identifier of the menu item. The MF_BYCOMMAND flag is the default if neither the MF_BYCOMMAND nor MF_BYPOSITION flag is specified.
MF_BYPOSITION	Indicates that the uPosition parameter gives the zero-based relative position of the menu item.

uIDNewItem

[in] Specifies either the identifier of the modified menu item or, if the uFlags parameter has the MF_POPUP flag set, a handle to the drop-down menu or submenu.

lpNewItem

[in] Pointer to the content of the changed menu item. The interpretation of this parameter depends on whether the uFlags parameter includes the MF_BITMAP, MF_OWNERDRAW, or MF_STRING flag.

Value	Meaning
MF_BITMAP	Contains a bitmap handle.
MF_OWNERDRAW	Contains a value supplied by an application that is used to maintain additional data related to the menu item. The value is in the itemData member of the structure pointed to by the lparam parameter of the WM_MEASUREITEM or WM_DRAWITEM messages sent when the menu item is created or its appearance is updated.
MF_STRING	Contains a pointer to a null-terminated string (the default).

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

If ModifyMenu replaces a menu item that opens a drop-down menu or submenu, the function destroys the old drop-down menu or submenu and frees the memory used by it.

In order for keyboard accelerators to work with bitmap or owner-drawn menu items, the owner of the menu must process the WM_MENUCHAR message. See Owner-Drawn Menus and the WM_MENUCHAR Message for more information.

The application must call the DrawMenuBar function whenever a menu changes, whether or not the menu is in a displayed window. To change the attributes of existing menu items, it is much faster to use the CheckMenuItem and EnableMenuItem functions.

The following list describes the flags that may be set in the uFlags parameter.

Value	Meaning
MF_BITMAP	Uses a bitmap as the menu item. The lpNewItem parameter contains a handle to the bitmap.
MF_BYCOMMAND	Indicates that the uPosition parameter specifies the identifier of the menu item (the default).
MF_BYPOSITION	Indicates that the uPosition parameter specifies the zero-based relative position of the new menu item.
MF_CHECKED	Places a check mark next to the item. If your application provides check-mark bitmaps (see the SetMenuItemBitmaps function), this flag displays a selected bitmap next to the menu item.
MF_DISABLED	Disables the menu item so that it cannot be selected, but this flag does not gray it.
MF_ENABLED	Enables the menu item so that it can be selected and restores it from its grayed state.
MF_GRAYED	Disables the menu item and grays it so that it cannot be selected.
MF_MENUBARBREAK	Functions the same as the MF_MENUBREAK flag for a menu bar. For a drop-down menu, submenu, or shortcut menu, the new column is separated from the old column by a vertical line.
MF_MENUBREAK	Places the item on a new line (for menu bars) or in a new column (for a drop-down menu, submenu, or shortcut menu) without separating columns.
MF_OWNERDRAW	Specifies that the item is an owner-drawn item. Before the menu is displayed for the first time, the window that owns the menu receives a WM_MEASUREITEM message to retrieve the width and height of the menu item. The WM_DRAWITEM message is then sent to the window procedure of the owner window whenever the appearance of the menu item must be updated.

MF_POPUP	Specifies that the menu item opens a drop-down menu or submenu. The uIDNewItem parameter specifies a handle to the drop-down menu or submenu. This flag is used to add a menu name to a menu bar or a menu item that opens a submenu to a drop-down menu, submenu, or shortcut menu.
MF_SEPARATOR	Draws a horizontal dividing line. This flag is used only in a drop-down menu, submenu, or shortcut menu. The line cannot be grayed, disabled, or highlighted. The lpNewItem and uIDNewItem parameters are ignored.
MF_STRING	Specifies that the menu item is a text string; the lpNewItem parameter is a pointer to the string.
MF_UNCHECKED	Does not place a check mark next to the item (the default). If your application supplies check-mark bitmaps (see the SetMenuItemBitmaps function), this flag displays a clear bitmap next to the menu item.

The following groups of flags cannot be used together:

- MF_BYCOMMAND and MF_BYPOSITION
- MF_DISABLED, MF_ENABLED, and MF_GRAYED
- MF_BITMAP, MF_STRING, MF_OWNERDRAW, and MF_SEPARATOR
- MF_MENUBARBREAK and MF_MENUBREAK
- MF_CHECKED and MF_UNCHECKED

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.309 MonitorFromPoint

The MonitorFromPoint function retrieves a handle to the display monitor that contains a specified point.

```
MonitorFromPoint: procedure
(
    pt              :POINT;
    dwFlags         :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__MonitorFromPoint@12" );
```

Parameters

pt

[in] A POINT structure that specifies the point of interest in virtual-screen coordinates.

dwFlags

[in] Determines the function's return value if the point is not contained within any display monitor.

This parameter can be one of the following values.

Value	Meaning
MONITOR_DEFAULTTONEAREST	Returns a handle to the display monitor that is nearest to the point.

MONITOR_DEFAULTTONUL Returns NULL.
L
MONITOR_DEFAULTTOPRIMReturns a handle to the primary display monitor.
ARY

Return Values

If the point is contained by a display monitor, the return value is an HMONITOR handle to that display monitor.

If the point is not contained by a display monitor, the return value depends on the value of dwFlags.

Requirements

Windows NT/2000: Requires Windows 2000 or later.

Windows 95/98: Requires Windows 98 or later.

3.310 MonitorFromRect

The MonitorFromRect function retrieves a handle to the display monitor that has the largest area of intersection with a specified rectangle.

```
MonitorFromRect: procedure
(
    var lprc        :RECT;
        dwflags     :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__MonitorFromRect@8" );
```

Parameters

lprc

[in] Pointer to a RECT structure that specifies the rectangle of interest in virtual-screen coordinates.

dwFlags

[in] Determines the function's return value if the rectangle does not intersect any display monitor.

This parameter can be one of the following values.

Value	Meaning
MONITOR_DEFAULTTONEAREST	Returns a handle to the display monitor that is nearest to the rectangle.
MONITOR_DEFAULTTONULL	Returns NULL.
MONITOR_DEFAULTTOPRIMARY	Returns a handle to the primary display monitor.

Return Values

If the rectangle intersects one or more display monitor rectangles, the return value is an HMONITOR handle to the display monitor that has the largest area of intersection with the rectangle.

If the rectangle does not intersect a display monitor, the return value depends on the value of dwFlags.

Requirements

Windows NT/2000: Requires Windows 2000 or later.

Windows 95/98: Requires Windows 98 or later.

3.311 MonitorFromWindow

The MonitorFromWindow function retrieves a handle to the display monitor that has the largest area of intersection with the bounding rectangle of a specified window.

```
MonitorFromWindow: procedure
(
    hwnd            :dword;
    dwFlags         :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__MonitorFromWindow@8" );
```

Parameters

hwnd

[in] Handle to the window of interest.

dwFlags

[in] Determines the function's return value if the window does not intersect any display monitor.

This parameter can be one of the following values.

Value	Meaning
MONITOR_DEFAULTTONEAREST	Returns a handle to the display monitor that is nearest to the window.
MONITOR_DEFAULTTONULL	Returns NULL.
MONITOR_DEFAULTTOPRIMARY	Returns a handle to the primary display monitor.

Return Values

If the window intersects one or more display monitor rectangles, the return value is an HMONITOR handle to the display monitor that has the largest area of intersection with the window.

If the window does not intersect a display monitor, the return value depends on the value of dwFlags.

Remarks

If the window is currently minimized, MonitorFromWindow uses the rectangle of the window before it was minimized.

Requirements

Windows NT/2000: Requires Windows 2000 or later.

Windows 95/98: Requires Windows 98 or later.

3.312 mouse_event

The mouse_event function synthesizes mouse motion and button clicks.

Windows NT/ 2000: This function has been superseded. Use SendInput instead.

```
mouse_event: procedure
(
    dwFlags     :dword;
    dx          :dword;
    dy          :dword;
```

```
    dwData      :dword;
    dwExtraInfo :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__mouse_event@20" );
```

Parameters

dwFlags

[in] Specifies various aspects of mouse motion and button clicking. This parameter can be certain combinations of the following values.

Value	Meaning
MOUSEEVENTF_ABSOLUTE	Specifies that the dx and dy parameters contain normalized absolute coordinates. If not set, those parameters contain relative data: the change in position since the last reported position. This flag can be set, or not set, regardless of what kind of mouse or mouse-like device, if any, is connected to the system. For further information about relative mouse motion, see the following Remarks section.
MOUSEEVENTF_MOVE	Specifies that movement occurred.
MOUSEEVENTF_LEFTDOWN	Specifies that the left button is down.
MOUSEEVENTF_LEFTUP	Specifies that the left button is up.
MOUSEEVENTF_RIGHTDOWN	Specifies that the right button is down.
MOUSEEVENTF_RIGHTUP	Specifies that the right button is up.
MOUSEEVENTF_MIDDLEDOWN	Specifies that the middle button is down.
MOUSEEVENTF_MIDDLEUP	Specifies that the middle button is up.
MOUSEEVENTF_WHEEL	Windows NT/ 2000: Specifies that the wheel has been moved, if the mouse has a wheel. The amount of movement is specified in dwData
MOUSEEVENTF_XDOWN	Windows 2000: Specifies that an X button was pressed.
MOUSEEVENTF_XUP	Windows 2000: Specifies that an X button was released.

The values that specify mouse button status are set to indicate changes in status, not ongoing conditions. For example, if the left mouse button is pressed and held down, MOUSEEVENTF_LEFTDOWN is set when the left button is first pressed, but not for subsequent motions. Similarly, MOUSEEVENTF_LEFTUP is set only when the button is first released.

You cannot specify both MOUSEEVENTF_WHEEL and either MOUSEEVENTF_XDOWN or MOUSEEVENTF_XUP simultaneously in the dwFlags parameter, because they both require use of the mouseData field.

dx

[in] Specifies the mouse's absolute position along the x-axis or its amount of motion since the last mouse event was generated, depending on the setting of MOUSEEVENTF_ABSOLUTE. Absolute data is specified as the mouse's actual x-coordinate; relative data is specified as the number of mickeys moved. A mickey is the amount that a mouse has to move for it to report that it has moved.

dy

[in] Specifies the mouse's absolute position along the y-axis or its amount of motion since the last mouse event was generated, depending on the setting of MOUSEEVENTF_ABSOLUTE. Absolute data is specified as the mouse's actual y-coordinate; relative data is specified as the number of mickeys moved.

dwData

[in] If dwFlags contains MOUSEEVENTF_WHEEL, then dwData specifies the amount of wheel movement. A positive value indicates that the wheel was rotated forward, away from the user; a negative value indicates that the wheel was rotated backward, toward the user. One wheel click is defined as WHEEL_DELTA, which is 120.

Windows 2000: If dwFlags contains MOUSEEVENTF_XDOWN or MOUSEEVENTF_XUP, then dwData specifies which X buttons were pressed or released. This value may be any combination of the following flags.

Value	Meaning
XBUTTON1	Set if the first X button was pressed or released.
XBUTTON2	Set if the second X button was pressed or released.

If dwFlags is not MOUSEEVENTF_WHEEL, MOUSEEVENTF_XDOWN, or MOUSEEVENTF_XUP, then dwData should be zero.

dwExtraInfo

[in] Specifies an additional value associated with the mouse event. An application calls GetMessageExtraInfo to obtain this extra information.

Return Values

This function has no return value.

Remarks

If the mouse has moved, indicated by MOUSEEVENTF_MOVE being set, dx and dy hold information about that motion. The information is specified as absolute or relative integer values.

If MOUSEEVENTF_ABSOLUTE value is specified, dx and dy contain normalized absolute coordinates between 0 and 65,535. The event procedure maps these coordinates onto the display surface. Coordinate (0,0) maps onto the upper-left corner of the display surface, (65535,65535) maps onto the lower-right corner.

If the MOUSEEVENTF_ABSOLUTE value is not specified, dx and dy specify relative motions from when the last mouse event was generated (the last reported position). Positive values mean the mouse moved right (or down); negative values mean the mouse moved left (or up).

Relative mouse motion is subject to the settings for mouse speed and acceleration level. An end user sets these values using the Mouse application in Control Panel. An application obtains and sets these values with the SystemParametersInfo function.

The system applies two tests to the specified relative mouse motion when applying acceleration. If the specified distance along either the x or y axis is greater than the first mouse threshold value, and the mouse acceleration level is not zero, the operating system doubles the distance. If the specified distance along either the x- or y-axis is greater than the second mouse threshold value, and the mouse acceleration level is equal to two, the operating system doubles the distance that resulted from applying the first threshold test. It is thus possible for the operating system to multiply relatively-specified mouse motion along the x- or y-axis by up to four times.

Once acceleration has been applied, the system scales the resultant value by the desired mouse speed. Mouse speed can range from 1 (slowest) to 20 (fastest) and represents how much the pointer moves based on the distance the mouse moves. The default value is 10, which results in no additional modification to the mouse motion.

The mouse_event function is used to synthesize mouse events by applications that need to do so. It is also used by applications that need to obtain more information from the mouse than its position and button state. For example, if a tablet manufacturer wants to pass pen-based information to its own applications, it can write a dynamic-link library (DLL) that communicates directly to the tablet hardware, obtains the extra information, and saves it in a queue. The DLL then calls mouse_event with the standard button and x/y position data, along with, in the dwExtraInfo parameter, some pointer or index to the queued extra information. When the application needs the extra information, it calls the DLL with the pointer or index stored in dwExtraInfo, and the DLL returns the extra information.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.313 MoveWindow

The MoveWindow function changes the position and dimensions of the specified window. For a top-level window, the position and dimensions are relative to the upper-left corner of the screen. For a child window, they are relative to the upper-left corner of the parent window's client area.

```
MoveWindow: procedure
(
    hWnd            :dword;
    X               :dword;
    Y               :dword;
    nWidth          :dword;
    nHeight         :dword;
    bRepaint        :boolean
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__MoveWindow@24" );
```

Parameters

hWnd

[in] Handle to the window.

X

[in] Specifies the new position of the left side of the window.

Y

[in] Specifies the new position of the top of the window.

nWidth

[in] Specifies the new width of the window.

nHeight

[in] Specifies the new height of the window.

bRepaint

[in] Specifies whether the window is to be repainted. If this parameter is TRUE, the window receives a WM_PAINT message. If the parameter is FALSE, no repainting of any kind occurs. This applies to the client area, the nonclient area (including the title bar and scroll bars), and any part of the parent window uncovered as a result of moving a child window.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

If the bRepaint parameter is TRUE, the system sends the WM_PAINT message to the window procedure immediately after moving the window (that is, the MoveWindow function calls the UpdateWindow function). If bRe-

paint is FALSE, the application must explicitly invalidate or redraw any parts of the window and parent window that need redrawing.

MoveWindow sends the WM_WINDOWPOSCHANGING, WM_WINDOWPOSCHANGED, WM_MOVE, WM_SIZE, and WM_NCCALCSIZE messages to the window.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.314 MsgWaitForMultipleObjects

The MsgWaitForMultipleObjects function returns when one of the following occurs:

- Either any one or all of the specified objects are in the signaled state. The objects can include input event objects, which you specify using the dwWakeMask parameter.
- The time-out interval elapses.

To enter an alertable wait state, use the MsgWaitForMultipleObjectsEx function.

Note MsgWaitForMultipleObjects does not return if there is unread input of the specified type in the message queue after the thread has called a function to check the queue. This is because functions such as PeekMessage, GetMessage, GetQueueStatus, and WaitMessage check the queue and then change the state information for the queue so that the input is no longer considered new. A subsequent call to MsgWaitForMultipleObjects will not return until new input of the specified type arrives. The existing unread input (received prior to the last time the thread checked the queue) is ignored.

```
MsgWaitForMultipleObjects: procedure
(
        nCount          :dword;
    var pHandles        :var;
        fWaitAll        :boolean;
        dwMilliseconds  :dword;
        dwWakeMask      :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__MsgWaitForMultipleObjects@20" );
```

Parameters

nCount

[in] Specifies the number of object handles in the array pointed to by pHandles. The maximum number of object handles is MAXIMUM_WAIT_OBJECTS minus one.

pHandles

[in] Pointer to an array of object handles. For a list of the object types whose handles can be specified, see the following Remarks section. The array can contain handles of objects of different types. It may not contain multiple copies of the same handle.

If one of these handles is closed while the wait is still pending, the function's behavior is undefined.

Windows NT/2000: The handles must have SYNCHRONIZE access. For more information, see Standard Access Rights.

Windows 95: No handle may be a duplicate of another handle created using DuplicateHandle.

fWaitAll

[in] Specifies the wait type. If TRUE, the function returns when the states of all objects in the pHandles array have been set to signaled and an input event has been received. If FALSE, the function returns when the state of any one of the objects is set to signaled or an input event has been received. In this case, the return value indicates the object whose state caused the function to return.

dwMilliseconds

[in] Specifies the time-out interval, in milliseconds. The function returns if the interval elapses, even if the criteria specified by the fWaitAll or dwWakeMask parameter have not been met. If dwMilliseconds is zero, the function tests the states of the specified objects and returns immediately. If dwMilliseconds is INFINITE, the function's time-out interval never elapses.

dwWakeMask

[in] Specifies input types for which an input event object handle will be added to the array of object handles. This parameter can be any combination of the following values.

Value	Meaning
QS_ALLEVENTS	An input, WM_TIMER, WM_PAINT, WM_HOTKEY, or posted message is in the queue. This value is a combination of QS_INPUT, QS_POSTMESSAGE, QS_TIMER, QS_PAINT, and QS_HOTKEY.
QS_ALLINPUT	Any message is in the queue. This value is a combination of QS_INPUT, QS_POSTMESSAGE, QS_TIMER, QS_PAINT, QS_HOTKEY, and QS_SENDMESSAGE.
QS_ALLPOSTMESSAGE	A posted message is in the queue. This value is cleared when you call GetMessage or Peek-Message, whether or not you are filtering messages.
QS_HOTKEY	A WM_HOTKEY message is in the queue.
QS_INPUT	An input message is in the queue. This value is a combination of QS_MOUSE and QS_KEY.
QS_KEY	A WM_KEYUP, WM_KEYDOWN, WM_SYSKEYUP, or WM_SYSKEYDOWN message is in the queue.
QS_MOUSE	A WM_MOUSEMOVE message or mouse-button message (WM_LBUTTONUP, WM_RBUTTONDOWN, and so on). This value is a combination of QS_MOUSEMOVE and QS_MOUSEBUTTON.
QS_MOUSEBUTTON	A mouse-button message (WM_LBUTTONUP, WM_RBUTTONDOWN, and so on).
QS_MOUSEMOVE	A WM_MOUSEMOVE message is in the queue.
QS_PAINT	A WM_PAINT message is in the queue.
QS_POSTMESSAGE	A posted message is in the queue. This value is cleared when you call GetMessage or Peek-Message without filtering messages.
QS_SENDMESSAGE	A message sent by another thread or application is in the queue.
QS_TIMER	A WM_TIMER message is in the queue.

Return Values

If the function succeeds, the return value indicates the event that caused the function to return. The successful return value is one of the following:

Value	Meaning

WAIT_OBJECT_0 to (WAIT_OBJECT_0 + nCount – 1)	If fWaitAll is TRUE, the return value indicates that the state of all specified objects is signaled. If fWaitAll is FALSE, the return value minus WAIT_OBJECT_0 indicates the pHandles array index of the object that satisfied the wait.
WAIT_OBJECT_0 + nCount	New input of the type specified in the dwWakeMask parameter is available in the thread's input queue. Functions such as PeekMessage, GetMessage, and WaitMessage mark messages in the queue as old messages. Therefore, after you call one of these functions, a subsequent call to will not return until new input of the specified type arrives. This value is also returned upon the occurrence of a system event that requires the thread's action, such as foreground activation. Therefore, can return even though no appropriate input is available and even if dwWaitMask is set to 0. If this occurs, call PeekMessage or GetMessage to process the system event before trying the call to again.
WAIT_ABANDONED_0 to (WAIT_ABANDONED_0 + nCount – 1)	If fWaitAll is TRUE, the return value indicates that the state of all specified objects is signaled and at least one of the objects is an abandoned mutex object. If fWaitAll is FALSE, the return value minus WAIT_ABANDONED_0 indicates the pHandles array index of an abandoned mutex object that satisfied the wait.
WAIT_TIMEOUT	The time-out interval elapsed and the conditions specified by the fWaitAll and dwWakeMask parameters were not satisfied.

If the function fails, the return value is WAIT_FAILED. To get extended error information, call GetLastError.

Remarks

The MsgWaitForMultipleObjects function determines whether the wait criteria have been met. If the criteria have not been met, the calling thread enters the wait state. It uses no processor time while waiting for the conditions of the wait criteria to be met.

When fWaitAll is TRUE, the function does not modify the states of the specified objects until the states of all objects have been set to signaled. For example, a mutex can be signaled, but the thread does not get ownership until the states of the other objects have also been set to signaled. In the meantime, some other thread may get ownership of the mutex, thereby setting its state to nonsignaled.

When fWaitAll is TRUE, the function's wait is completed only when the states of all objects have been set to signaled and an input event has been received. Therefore, setting fWaitAll to TRUE prevents input from being processed until the state of all objects in the pHandles array have been set to signaled. For this reason, if you set fWaitAll to TRUE, you should use a short timeout value in dwMilliseconds. If you have a thread that creates windows waiting for all objects in the pHandles array, including input events specified by dwWakeMask, with no timeout interval, the system will deadlock. This is because threads that create windows must process messages. DDE sends message to all windows in the system. Therefore, if a thread creates windows, do not set the fWaitAll parameter to TRUE in calls to MsgWaitForMultipleObjects made from that thread.

The function modifies the state of some types of synchronization objects. Modification occurs only for the object or objects whose signaled state caused the function to return. For example, the count of a semaphore object is decreased by one. When fWaitAll is FALSE, and multiple objects are in the signaled state, the function chooses one of the objects to satisfy the wait; the states of the objects not selected are unaffected.

The MsgWaitForMultipleObjects function can specify handles of any of the following object types in the pHandles array:

- Change notification

- Console input
- Event
- Job
- Mutex
- Process
- Semaphore
- Thread
- Waitable timer

For more information, see Synchronization Objects.

The QS_ALLPOSTMESSAGE and QS_POSTMESSAGE flags differ in when they are cleared. QS_POSTMESSAGE is cleared when you call GetMessage or PeekMessage, whether or not you are filtering messages. QS_ALLPOSTMESSAGE is cleared when you call GetMessage or PeekMessage without filtering messages (wMsgFilterMin and wMsgFilterMax are 0). This can be useful when you call PeekMessage multiple times to get messages in different ranges.

Example

For an example that uses MsgWaitForMultipleObjects, see Waiting in a Message Loop.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.315 MsgWaitForMultipleObjectsEx

The MsgWaitForMultipleObjectsEx function returns when one of the following occurs:

- Either any one or all of the specified objects are in the signaled state. The array of objects can include input event objects, which you specify using the dwWakeMask parameter.
- An I/O completion routine or asynchronous procedure call (APC) is queued to the thread.
- The time-out interval elapses.

Note MsgWaitForMultipleObjectsEx does not return if there is unread input of the specified type in the message queue after the thread has called a function to check the queue, unless you use the MWMO_INPUTAVAILABLE flag. This is because functions such as PeekMessage, GetMessage, GetQueueStatus, and WaitMessage check the queue and then change the state information for the queue so that the input is no longer considered new. A subsequent call to MsgWaitForMultipleObjectsEx will not return until new input of the specified type arrives, unless you use the MWMO_INPUTAVAILABLE flag. If this flag is not used, the existing unread input (received prior to the last time the thread checked the queue) is ignored.

```
MsgWaitForMultipleObjectsEx: procedure
(
        nCount          :dword;
    var pHandles        :var;
        fWaitAll        :boolean;
        dwMilliseconds  :dword;
        dwWakeMask      :dword;
        dwFlags         :dword
);
    @stdcall;
```

```
@returns( "eax" );
@external( "__imp__MsgWaitForMultipleObjectsEx@20" );
```

Parameters

nCount

[in] Specifies the number of object handles in the array pointed to by pHandles. The maximum number of object handles is MAXIMUM_WAIT_OBJECTS minus one.

pHandles

[in] Pointer to an array of object handles. For a list of the object types whose handles you can specify, see the Remarks section later in this topic. The array can contain handles to multiple types of objects. It may not contain multiple copies of the same handle.

If one of these handles is closed while the wait is still pending, the function's behavior is undefined.

Windows NT/2000: The handles must have SYNCHRONIZE access. For more information, see Standard Access Rights.

dwMilliseconds

[in] Specifies the time-out interval, in milliseconds. The function returns if the interval elapses, even if the conditions specified by the dwWakeMask and dwFlags parameters are not met. If dwMilliseconds is zero, the function tests the states of the specified objects and returns immediately. If dwMilliseconds is INFINITE, the function's time-out interval never elapses.

dwWakeMask

[in] Specifies input types for which an input event object handle will be added to the array of object handles. This parameter can be any combination of the following values:

Value	Meaning
QS_ALLEVENTS	An input, WM_TIMER, WM_PAINT, WM_HOTKEY, or posted message is in the queue. This value is a combination of QS_INPUT, QS_POSTMESSAGE, QS_TIMER, QS_PAINT, and QS_HOTKEY.
QS_ALLINPUT	Any message is in the queue. This value is a combination of QS_INPUT, QS_POSTMESSAGE, QS_TIMER, QS_PAINT, QS_HOTKEY, and QS_SENDMESSAGE.
QS_ALLPOSTMESSAGE	A posted message (other than those listed here) is in the queue.
QS_HOTKEY	A WM_HOTKEY message is in the queue.
QS_INPUT	An input message is in the queue. This value is a combination of QS_MOUSE and QS_KEY.
QS_KEY	A WM_KEYUP, WM_KEYDOWN, WM_SYSKEYUP, or WM_SYSKEYDOWN message is in the queue.
QS_MOUSE	A WM_MOUSEMOVE message or mouse-button message (WM_LBUTTONUP, WM_RBUTTONDOWN, and so on) is in the queue. This value is a combination of QS_MOUSEMOVE and QS_MOUSEBUTTON.
QS_MOUSEBUTTON	A mouse-button message (WM_LBUTTONUP, WM_RBUTTONDOWN, and so on) is in the queue.
QS_MOUSEMOVE	A WM_MOUSEMOVE message is in the queue.

QS_PAINT	A WM_PAINT message is in the queue.
QS_POSTMESSAGE	A posted message (other than those just listed) is in the queue.
QS_SENDMESSAGE	A message sent by another thread or application is in the queue.
QS_TIMER	A WM_TIMER message is in the queue.

dwFlags

[in] Specifies the wait type. This parameter can be any combination of the following values:

Value	Meaning
0	The function returns when any one of the objects is signaled. The return value indicates the object whose state caused the function to return.
MWMO_WAITALL	The function returns when all objects in the pHandles array are signaled and an input event has been received, all at the same time.
MWMO_ALERTABLE	The function also returns if an APC has been queued to the thread with QueueUserAPC.
MWMO_INPUTAVAILABLE	Windows 98, Windows 2000: The function returns if input exists for the queue, even if the input has been seen (but not removed) using a call to another function, such as PeekMessage.

Return Values

If the function succeeds, the return value indicates the event that caused the function to return. The successful return value is one of the following:

Value	Meaning
WAIT_OBJECT_0 to (WAIT_OBJECT_0 + nCount - 1)	If the MWMO_WAITALL flag is used, the return value indicates that the state of all specified objects is signaled. Otherwise, the return value minus WAIT_OBJECT_0 indicates the pHandles array index of the object that caused the function to return.
WAIT_OBJECT_0 + nCount	New input of the type specified in the dwWakeMask parameter is available in the thread's input queue. Functions such as PeekMessage, GetMessage, GetQueueStatus, and WaitMessage mark messages in the queue as old messages. Therefore, after you call one of these functions, a subsequent call to will not return until new input of the specified type arrives. This value is also returned upon the occurrence of a system event that requires the thread's action, such as foreground activation. Therefore, can return even though no appropriate input is available and even if dwWaitMask is set to 0. If this occurs, call PeekMessage or GetMessage to process the system event before trying the call to again.

WAIT_ABANDONED_0 to (WAIT_ABANDONED_0 + nCount - 1)	If the MWMO_WAITALL flag is used, the return value indicates that the state of all specified objects is signaled and at least one of the objects is an abandoned mutex object. Otherwise, the return value minus WAIT_ABANDONED_0 indicates the pHandles array index of an abandoned mutex object that caused the function to return.
WAIT_IO_COMPLETION	The wait was ended by a user-mode asynchronous procedure call (APC) queued to the thread.
WAIT_TIMEOUT	The time-out interval elapsed, but the conditions specified by the dwFlags and dwWakeMask parameters were not met.

If the function fails, the return value is WAIT_FAILED. To get extended error information, call GetLastError.

Remarks

The MsgWaitForMultipleObjectsEx function determines whether the conditions specified by dwWakeMask and dwFlags have been met. If the conditions have not been met, the calling thread enters the wait state. It uses no processor time while waiting for one of the conditions to be met or for the time-out interval to elapse.

The function modifies the state of some types of synchronization objects. Modification occurs only for the object or objects whose signaled state caused the function to return. For example, the system decreases the count of a semaphore object by one. When dwFlags is zero, and multiple objects are in the signaled state, the function chooses one of the objects to satisfy the wait; the states of the objects not selected are unaffected.

The MsgWaitForMultipleObjectsEx function can specify handles of any of the following object types in the pHandles array:

- Change notification
- Console input
- Event
- Job
- Mutex
- Process
- Semaphore
- Thread
- Waitable timer

For more information, see Synchronization Objects.

The QS_ALLPOSTMESSAGE and QS_POSTMESSAGE flags differ in when they are cleared. QS_POSTMESSAGE is cleared when you call GetMessage or PeekMessage, whether or not you are filtering messages. QS_ALLPOSTMESSAGE is cleared when you call GetMessage or PeekMessage without filtering messages (wMsgFilterMin and wMsgFilterMax are 0). This can be useful when you call PeekMessage multiple times to get messages in different ranges.

Requirements

Windows NT/2000: Requires Windows NT 4.0 or later.

Windows 95/98: Requires Windows 98 or later.

3.316 OemKeyScan

The OemKeyScan function maps OEM ASCII codes 0 through 0x0FF into the OEM scan codes and shift states. The function provides information that allows a program to send OEM text to another program by simulating keyboard input.

```
OemKeyScan: procedure
(
    wOemChar    :word
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__OemKeyScan@4" );
```

Parameters

wOemChar

[in] Specifies the ASCII value of the OEM character.

Return Values

The low-order word of the return value contains the scan code of the OEM character, and the high-order word contains the shift state, which can be a combination of the following bits.

Bit	Meaning
1	Either SHIFT key is pressed.
2	Either CTRL key is pressed.
4	Either ALT key is pressed.
8	The Hankaku key is pressed.
16	Reserved (defined by the keyboard layout driver).
32	Reserved (defined by the keyboard layout driver).

If the character cannot be produced by a single keystroke using the current keyboard layout, the return value is –1.

Remarks

This function does not provide translations for characters that require CTRL+ALT or dead keys. Characters not translated by this function must be copied by simulating input using the ALT+ keypad mechanism. The NUM-LOCK key must be off.

This function does not provide translations for characters that cannot be typed with one keystroke using the current keyboard layout, such as characters with diacritics requiring dead keys. Characters not translated by this function may be simulated using the ALT+ keypad mechanism. The NUMLOCK key must be on.

This function is implemented using the VkKeyScan function.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.317 OemToChar

The OemToChar function translates a string from the OEM-defined character set into either an ANSI or a wide-character string.

```
OemToChar: procedure
```

```
(
        lpszSrc     :string;
    var lpszDst     :var
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__OemToCharA@8" );
```

Parameters

lpszSrc

[in] Pointer to a null-terminated string of characters from the OEM-defined character set.

lpszDst

[out] Pointer to the buffer for the translated string. If the OemToChar function is being used as an ANSI function, the string can be translated in place by setting the lpszDst parameter to the same address as the lpszSrc parameter. This cannot be done if OemToChar is being used as a wide-character function.

Return Values

The return value is always nonzero.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.318 OemToCharBuff

The OemToCharBuff function translates a specified number of characters in a string from the OEM-defined character set into either an ANSI or a wide-character string.

```
OemToCharBuff: procedure
(
        lpszSrc         :string;
    var lpszDst         :var;
        cchDstLength    :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__OemToCharBuffA@12" );
```

Parameters

lpszSrc

[in] Pointer to a buffer containing one or more characters from the OEM-defined character set.

lpszDst

[out] Pointer to the buffer for the translated string. If the OemToCharBuff function is being used as an ANSI function, the string can be translated in place by setting the lpszDst parameter to the same address as the lpszSrc parameter. This cannot be done if the OemToCharBuff function is being used as a wide-character function.

cchDstLength

[in] Specifies the number of TCHARs to translate in the buffer identified by the lpszSrc parameter. This refers to bytes for ANSI versions of the function or characters for Unicode versions.

Return Values

The return value is always nonzero.

Remarks

Unlike the OemToChar function, the OemToCharBuff function does not stop converting characters when it encounters a null character in the buffer pointed to by lpszSrc. The OemToCharBuff function converts all cchDstLength characters.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.319 OffsetRect

The OffsetRect function moves the specified rectangle by the specified offsets.

```
OffsetRect: procedure
(
    var lprc        :RECT;
        dx          :dword;
        dy          :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__OffsetRect@12" );
```

Parameters

lprc

[in/out] Pointer to a RECT structure that contains the logical coordinates of the rectangle to be moved.

dx

[in] Specifies the amount to move the rectangle left or right. This parameter must be a negative value to move the rectangle to the left.

dy

[in] Specifies the amount to move the rectangle up or down. This parameter must be a negative value to move the rectangle up.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Windows NT/ 2000: To get extended error information, call GetLastError.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.320 OpenClipboard

The OpenClipboard function opens the clipboard for examination and prevents other applications from modifying the clipboard content.

```
OpenClipboard: procedure
(
    hWndNewOwner    :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__OpenClipboard@4" );
```

Parameters

hWndNewOwner

[in] Handle to the window to be associated with the open clipboard. If this parameter is NULL, the open clipboard is associated with the current task.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

OpenClipboard fails if another window has the clipboard open.

An application should call the CloseClipboard function after every successful call to OpenClipboard.

The window identified by the hWndNewOwner parameter does not become the clipboard owner unless the EmptyClipboard function is called.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.321 OpenDesktop

The OpenDesktop function retrieves a handle to an existing desktop. A desktop is a secure object contained within a window station object. A desktop has a logical display surface and contains windows, menus and hooks.

```
HDESK OpenDesktop(
  LPTSTR lpszDesktop,         // desktop name
  DWORD dwFlags,              // interaction option
  BOOL fInherit,              // inheritance option
  ACCESS_MASK dwDesiredAccess  // handle access
);
```

Parameters

lpszDesktop

[in] Pointer to null-terminated string specifying the name of the desktop to be opened. Desktop names are case-insensitive.

dwFlags

[in] Specifies how the calling application will cooperate with other applications on the desktop. This parameter can be zero or the following value.

Value	Description
DF_ALLOWOTHERACCOUNTHOOK	Allows processes running in other accounts on the desktop to set hooks in this process.

fInherit

[in] Specifies whether the returned handle is inherited when a new process is created. If this value is TRUE, new processes will inherit the handle.

dwDesiredAccess

[in] Specifies the access rights the returned handle has to the desktop. This parameter can include any of the standard access rights, such as READ_CONTROL or WRITE_DAC, and a combination of the following desktop-specific access rights.

Value	Description
DESKTOP_CREATEMENU	Required to create a menu on the desktop.
DESKTOP_CREATEWINDOW	Required to create a window on the desktop.
DESKTOP_ENUMERATE	Required for the desktop to be enumerated.
DESKTOP_HOOKCONTROL	Required to establish any of the window hooks.
DESKTOP_JOURNALPLAYBACK	Required to perform journal playback on the desktop.
DESKTOP_JOURNALRECORD	Required to perform journal recording on the desktop.
DESKTOP_READOBJECTS	Required to read objects on the desktop.
DESKTOP_SWITCHDESKTOP	Required to activate the desktop using SwitchDesktop.
DESKTOP_WRITEOBJECTS	Required to write objects on the desktop.

Return Values

If the function succeeds, the return value is a handle to the opened desktop. When you are finished using the handle, call the CloseDesktop function to close it.

If the function fails, the return value is NULL. To get extended error information, call GetLastError.

Remarks

The calling process must have an associated window station, either assigned by the system at process creation time or set by the SetProcessWindowStation function.

If the dwDesiredAccess parameter specifies the READ_CONTROL, WRITE_DAC, or WRITE_OWNER standard access rights to access the security descriptor of the desktop object, you must also request the DESKTOP_READOBJECTS and DESKTOP_WRITEOBJECTS access rights.

Requirements

Windows NT/2000: Requires Windows NT 3.51 or later.

Windows 95/98: Unsupported.

3.322 OpenIcon

The OpenIcon function restores a minimized (iconic) window to its previous size and position; it then activates the window.

```
OpenIcon: procedure
```

```
(
    hWnd            :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__OpenIcon@4" );
```

Parameters

hWnd

[in] Handle to the window to be restored and activated.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

OpenIcon sends a WM_QUERYOPEN message to the given window.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.323 OpenInputDesktop

The OpenInputDesktop function retrieves a handle to the desktop that receives user input. The input desktop is a desktop on the window station associated with the logged-on user.

```
OpenInputDesktop: procedure
(
    dwFlags         :dword;
    fInherit        :boolean;
    dwDesiredAccess :ACCESS_MASK
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__OpenInputDesktop@12" );
```

Parameters

dwFlags

[in] Specifies how the calling application will cooperate with other applications on the desktop. This parameter can be zero or the following value.

Value	Description
DF_ALLOWOTHERACCOUNTHOOK	Allows processes running in other accounts on the desktop to set hooks in this process.

fInherit

[in] Specifies whether the returned handle is inherited when a new process is created. If this value is TRUE, new processes will inherit the handle.

dwDesiredAccess

[in] Specifies the access rights the returned handle has to the desktop. This parameter can include any of the

standard access rights, such as READ_CONTROL or WRITE_DAC, and a combination of the following desktop-specific access rights.

Value	Description
DESKTOP_CREATEMENU	Required to create a menu on the desktop.
DESKTOP_CREATEWINDOW	Required to create a window on the desktop.
DESKTOP_ENUMERATE	Required for the desktop to be enumerated.
DESKTOP_HOOKCONTROL	Required to establish any of the window hooks.
DESKTOP_JOURNALPLAYBACK	Required to perform journal playback on the desktop.
DESKTOP_JOURNALRECORD	Required to perform journal recording on the desktop.
DESKTOP_READOBJECTS	Required to read objects on the desktop.
DESKTOP_SWITCHDESKTOP	Required to activate the desktop using Switch-Desktop.
DESKTOP_WRITEOBJECTS	Required to write objects on the desktop.

Return Values

If the function succeeds, the return value is a handle to the desktop that receives user input. When you are finished using the handle, call the CloseDesktop function to close it.

If the function fails, the return value is NULL. To get extended error information, call GetLastError.

Remarks

The calling process must have an associated window station, either assigned by the system at process creation time or set by SetProcessWindowStation. The window station associated with the calling process must be capable of receiving input.

An application can use the SwitchDesktop function to change the input desktop.

If the dwDesiredAccess parameter specifies the READ_CONTROL, WRITE_DAC, or WRITE_OWNER standard access rights to access the security descriptor of the desktop object, you must also request the DESKTOP_READOBJECTS and DESKTOP_WRITEOBJECTS access rights.

Requirements

Windows NT/2000: Requires Windows NT 3.51 or later.

Windows 95/98: Unsupported.

3.324 OpenWindowStation

The OpenWindowStation function retrieves a handle to an existing window station.

```
OpenWindowStation: procedure
(
    lpszWinSta      :string;
    fInherit        :boolean;
    dwDesiredAccess :ACCESS_MASK
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__OpenWindowStationA@12" );
```

Parameters

lpszWinSta

[in] Pointer to a null-terminated string specifying the name of the window station to be opened. Window sta-

tion names are case-insensitive.

fInherit

[in] Specifies whether the returned handle is inherited when a new process is created. If this value is TRUE, new processes will inherit the handle.

dwDesiredAccess

[in] Specifies the type of access to the window station. This parameter can be one or more of the following values.

Value	Description
WINSTA_ACCESSCLIPBOARD	Required to use the clipboard.
WINSTA_ACCESSGLOBALATOMS	Required to manipulate global atoms.
WINSTA_CREATEDESKTOP	Required to create new desktop objects on the window station.
WINSTA_ENUMDESKTOPS	Required to enumerate existing desktop objects.
WINSTA_ENUMERATE	Required for the window station to be enumerated.
WINSTA_EXITWINDOWS	Required to successfully call the ExitWindows and ExitWindowsEx functions.
WINSTA_READATTRIBUTES	Required to read the attributes of a window station object.
WINSTA_READSCREEN	Required to access screen contents.
WINSTA_WRITEATTRIBUTES	Required to modify the attributes of a window station object.

Return Values

If the function succeeds, the return value is the handle to the specified window station.

If the function fails, the return value is NULL. To get extended error information, call GetLastError.

Remarks

After you are done with the HWINSTA handle, you must call CloseWindowStation to free the handle.

Requirements

Windows NT/2000: Requires Windows NT 3.51 or later.

Windows 95/98: Unsupported.

3.325 PackDDElParam

The PackDDElParam function packs a DDE lParam value into an internal structure used for sharing DDE data between processes.

```
PackDDElParam: procedure
(
    msg             :dword;
    uiLo            :dword;
    uiHi        :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__PackDDElParam@12" );
```

Parameters

msg

[in] Specifies the DDE message to be posted.

uiLo

[in] Specifies a value that corresponds to the 16-bit Windows low-order word of an lParam parameter for the DDE message being posted.

uiHi

[in] Specifies a value that corresponds to the 16-bit Windows high-order word of an lParam parameter for the DDE message being posted.

Return Values

The return value is the lParam value.

Remarks

The return value must be posted as the lParam parameter of a DDE message; it must not be used for any other purpose. After the application posts a return value, it need not perform any action to dispose of the lParam parameter.

PackDDElParam eases the porting of 16-bit Windows-based applications to Win32-based applications.

An application should call this function only for posted DDE messages.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.326 PaintDesktop

The PaintDesktop function fills the clipping region in the specified device context with the desktop pattern or wallpaper. The function is provided primarily for shell desktops.

```
PaintDesktop: procedure
(
    hdc             :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__PaintDesktop@4" );
```

Parameters

hdc

[in] Handle to the device context.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Windows NT/ 2000: To get extended error information, call GetLastError.

Requirements

Windows NT/2000: Requires Windows NT 4.0 or later.

Windows 95/98: Requires Windows 95 or later.

3.327 PeekMessage

The PeekMessage function dispatches incoming sent messages, checks the thread message queue for a posted message, and retrieves the message (if any exist).

```
PeekMessage: procedure
(
    var lpMsg       :MSG;
    hWnd            :dword;
    wMsgFilterMin   :dword;
    wMsgFilterMax   :dword;
    wRemoveMsg      :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__PeekMessageA@20" );
```

Parameters

lpMsg

[out] Pointer to an MSG structure that receives message information.

hWnd

[in] Handle to the window whose messages are to be examined. The window must belong to the current thread.

If hWnd is NULL, PeekMessage retrieves messages for any window that belongs to the current thread. If hWnd is INVALID_HANDLE_VALUE, PeekMessage retrieves messages whose hWnd value is NULL, as posted by the PostThreadMessage function.

wMsgFilterMin

[in] Specifies the value of the first message in the range of messages to be examined. Use WM_KEYFIRST to specify the first keyboard message or WM_MOUSEFIRST to specify the first mouse message.

If wMsgFilterMin and wMsgFilterMax are both zero, PeekMessage returns all available messages (that is, no range filtering is performed).

wMsgFilterMax

[in] Specifies the value of the last message in the range of messages to be examined. Use WM_KEYLAST to specify the first keyboard message or WM_MOUSELAST to specify the last mouse message.

If wMsgFilterMin and wMsgFilterMax are both zero, PeekMessage returns all available messages (that is, no range filtering is performed).

wRemoveMsg

[in] Specifies how messages are handled. This parameter can be one of the following values.

Value	Meaning
PM_NOREMOVE	Messages are not removed from the queue after processing by .
PM_REMOVE	Messages are removed from the queue after processing by .

You can optionally combine the value PM_NOYIELD with either PM_NOREMOVE or PM_REMOVE. This flag prevents the system from releasing any thread that is waiting for the caller to go idle (see WaitForInputI-

dle).
By default, all message types are processed. To specify that only certain message should be processed, specify one of more of the following values.

Value	Meaning
PM_QS_INPUT	Windows 98, Windows 2000: Process mouse and keyboard messages.
PM_QS_PAINT	Windows 98, Windows 2000: Process paint messages.
PM_QS_POSTMESSAGE	Windows 98, Windows 2000: Process all posted messages, including timers and hotkeys.
PM_QS_SENDMESSAGE	Windows 98, Windows 2000: Process all sent messages.

Return Values

If a message is available, the return value is nonzero.

If no messages are available, the return value is zero.

Remarks

PeekMessage retrieves messages associated with the window identified by the hWnd parameter or any of its children as specified by the IsChild function, and within the range of message values given by the wMsgFilterMin and wMsgFilterMax parameters. Note that an application can only use the low word in the wMsgFilterMin and wMsgFilterMax parameters; the high word is reserved for the system.

Note that PeekMessage always retrieves WM_QUIT messages, no matter which values you specify for wMsgFilterMin and wMsgFilterMax.

During this call, the system delivers pending messages that were sent to windows owned by the calling thread using the SendMessage, SendMessageCallback, SendMessageTimeout, or SendNotifyMessage function. The system may also process internal events. Messages are processed in the following order:

- Sent messages
- Posted messages
- Input (hardware) messages and system internal events
- Sent messages (again)
- WM_PAINT messages
- WM_TIMER messages

To retrieve input messages before posted messages, use the wMsgFilterMin and wMsgFilterMax parameters.

The PeekMessage function normally does not remove WM_PAINT messages from the queue. WM_PAINT messages remain in the queue until they are processed. However, if a WM_PAINT message has a null update region, PeekMessage does remove it from the queue.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.328 PostMessage

The PostMessage function places (posts) a message in the message queue associated with the thread that created the specified window and returns without waiting for the thread to process the message.

To post a message in the message queue associate with a thread, use the PostThreadMessage function.

```
PostMessage: procedure
(
    hWnd            :dword;
    _Msg            :dword;
    wParam          :dword;
    lParam          :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__PostMessageA@16" );
```

Parameters

hWnd

[in] Handle to the window whose window procedure is to receive the message. The following values have special meanings.

Value	Meaning
HWND_BROADCAST	The message is posted to all top-level windows in the system, including disabled or invisible unowned windows, overlapped windows, and pop-up windows. The message is not posted to child windows.
NULL	The function behaves like a call to PostThreadMessage with the dwThreadId parameter set to the identifier of the current thread.

Msg

[in] Specifies the message to be posted.

wParam

[in] Specifies additional message-specific information.

lParam

[in] Specifies additional message-specific information.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

Messages in a message queue are retrieved by calls to the GetMessage or PeekMessage function.

Applications that need to communicate using HWND_BROADCAST should use the RegisterWindowMessage function to obtain a unique message for inter-application communication.

If you send a message in the range below WM_USER to the asynchronous message functions (PostMessage, SendNotifyMessage, and SendMessageCallback), its message parameters can not include pointers. Otherwise, the operation will fail. The functions will return before the receiving thread has had a chance to process the message and the sender will free the memory before it is used.

Do not post the WM_QUIT message using PostMessage; use the PostQuitMessage function.

Windows 2000: There is a limit of 10,000 posted messages per message queue. This limit should be sufficiently large. If your application exceeds the limit, it should be redesigned to avoid consuming so many system resources. To adjust this limit, modify the following registry key:

HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Windows\USERPostMessageLimit.

The minimum acceptable value is 4000.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.329 PostQuitMessage

The PostQuitMessage function indicates to the system that a thread has made a request to terminate (quit). It is typically used in response to a WM_DESTROY message.

```
PostQuitMessage: procedure
(
    nExitCode       :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__PostQuitMessage@4" );
```

Parameters

nExitCode

[in] Specifies an application exit code. This value is used as the wParam parameter of the WM_QUIT message.

Return Values

This function does not return a value.

Remarks

The PostQuitMessage function posts a WM_QUIT message to the thread's message queue and returns immediately; the function simply indicates to the system that the thread is requesting to quit at some time in the future.

When the thread retrieves the WM_QUIT message from its message queue, it should exit its message loop and return control to the system. The exit value returned to the system must be the wParam parameter of the WM_QUIT message.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.330 PostThreadMessage

The PostThreadMessage function posts a message to the message queue of the specified thread. It returns without waiting for the thread to process the message.

```
PostThreadMessage: procedure
(
    idThread        :dword;
    _Msg            :dword;
    _wParam         :dword;
    _lParam         :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__PostThreadMessageA@16" );
```

Parameters

idThread

[in] Identifier of the thread to which the message is to be posted.

The function fails if the specified thread does not have a message queue. The system creates a thread's message queue when the thread makes its first call to one of the Win32 User or GDI functions. For more information, see the Remarks section.

Windows 2000: This thread must either belong to the same desktop as the calling thread or to a process with the same LUID. Otherwise, the function fails and returns ERROR_INVALID_THREAD_ID.

Msg

[in] Specifies the type of message to be posted.

wParam

[in] Specifies additional message-specific information.

lParam

[in] Specifies additional message-specific information.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError. GetLastError returns ERROR_INVALID_THREAD_ID if idThread is not a valid thread identifier, or if the thread specified by idThread does not have a message queue.

Remarks

The thread to which the message is posted must have created a message queue, or else the call to PostThreadMessage fails. Use one of the following methods to handle this situation:

- Call PostThreadMessage. If it fails, call the Sleep function and call PostThreadMessage again. Repeat until PostThreadMessage succeeds.
- Create an event object, then create the thread. Use the WaitForSingleObject function to wait for the event to be set to the signaled state before calling PostThreadMessage. In the thread to which the message will be posted, call PeekMessage(&msg, NULL, WM_USER, WM_USER, PM_NOREMOVE) to force the system to create the message queue. Set the event, to indicate that the thread is ready to receive posted messages.

The thread to which the message is posted retrieves the message by calling the GetMessage or PeekMessage function. The hwnd member of the returned MSG structure is NULL.

Messages sent by PostThreadMessage are not associated with a window. Messages that are not associated with a window cannot be dispatched by the DispatchMessage function. Therefore, if the recipient thread is in a modal loop (as used by MessageBox or DialogBox), the messages will be lost. To intercept thread messages while in a modal loop, use a thread-specific hook.

Windows 2000: There is a limit of 10,000 posted messages per message queue. This limit should be sufficiently large. If your application exceeds the limit, it should be redesigned to avoid consuming so many system resources. To adjust this limit, modify the following registry key:

HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Windows\USERPostMessageLimit.

The minimum acceptable value is 4000.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.331 PtInRect

The PtInRect function determines whether the specified point lies within the specified rectangle. A point is within a rectangle if it lies on the left or top side or is within all four sides. A point on the right or bottom side is considered outside the rectangle.

```
PtInRect: procedure
(
    var lprc        :RECT;
        pt          :POINT
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__PtInRect@12" );
```

Parameters

lprc

[in] Pointer to a RECT structure that contains the specified rectangle.

pt

[in] Specifies a POINT structure that contains the specified point.

Return Values

If the specified point lies within the rectangle, the return value is nonzero.

If the specified point does not lie within the rectangle, the return value is zero.

Remarks

The rectangle must be normalized before PtInRect is called. That is, lprc.right must be greater than lprc.left and lprc.bottom must be greater than lprc.top. If the rectangle is not normalized, a point is never considered inside of the rectangle.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.332 RealChildWindowFromPoint

The RealChildWindowFromPoint function retrieves a handle to the child window at the specified point.

```
RealChildWindowFromPoint: procedure
(
    hwndParent              :dword;
    ptParentClientCoords    :POINT
);
    @stdcall;
    @returns( "eax" );
```

```
    @external( "__imp__RealChildWindowFromPoint@12" );
```

Parameters

hwndParent

[in] Handle to the window whose child is to be retrieved.

ptParentClientCoords

[in] Specifies a POINT structure that defines the client coordinates of the point to be checked.

Return Values

The return value is a handle to the child window that contains the specified point.

Requirements

Windows NT/2000: Requires Windows NT 4.0 SP4 or later.

Windows 95/98: Requires Windows 98 or later.

3.333 RealGetWindowClass

The RealGetWindowClass function retrieves a string that specifies the window type.

```
RealGetWindowClass: procedure
(
        hwnd        :dword;
    var pszType     :var;
        cchType     :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__RealGetWindowClassA@12" );

RealGetWindowClassW: procedure
(
        hwnd        :dword;
    var pszType     :var;
        cchType     :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__RealGetWindowClass@12" );
```

Parameters

hwnd

[in] Handle to the window whose type will be retrieved.

pszType

[out] Pointer to a string that receives the window type.

cchType

[in] Specifies the length, in TCHARs, of the buffer pointed to by the pszType parameter.

Return Values

If the function succeeds, the return value is the number of TCHARs copied to the specified buffer.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Requirements

Windows NT/2000: Requires Windows NT 4.0 SP4 or later.

Windows 95/98: Requires Windows 98 or later.

3.334 RedrawWindow

The RedrawWindow function updates the specified rectangle or region in a window's client area.

```
RedrawWindow: procedure
(
        hWnd        :dword;
    var lprcUpdate  :RECT;
        hrgnUpdate  :dword;
        flags       :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__RedrawWindow@16" );
```

Parameters

hWnd

[in] Handle to the window to be redrawn. If this parameter is NULL, the desktop window is updated.

lprcUpdate

[in] Pointer to a RECT structure containing the coordinates of the update rectangle. This parameter is ignored if the hrgnUpdate parameter identifies a region.

hrgnUpdate

[in] Handle to the update region. If both the hrgnUpdate and lprcUpdate parameters are NULL, the entire client area is added to the update region.

flags

[in] Specifies one or more redraw flags. This parameter can be used to invalidate or validate a window, control repainting, and control which windows are affected by RedrawWindow.

The following flags are used to invalidate the window.

Flag (invalidation)	Description
RDW_ERASE	Causes the window to receive a WM_ERASEBKGND message when the window is repainted. The RDW_INVALIDATE flag must also be specified; otherwise, RDW_ERASE has no effect.
RDW_FRAME	Causes any part of the nonclient area of the window that intersects the update region to receive a WM_NCPAINT message. The RDW_INVALIDATE flag must also be specified; otherwise, RDW_FRAME has no effect. The WM_NCPAINT message is typically not sent during the execution of unless either RDW_UPDATENOW or RDW_ERASENOW is specified.
RDW_INTERNALPAINT	Causes a WM_PAINT message to be posted to the window regardless of whether any portion of the window is invalid.

RDW_INVALIDATE	Invalidates lprcUpdate or hrgnUpdate (only one may be non-NULL). If both are NULL, the entire window is invalidated.

The following flags are used to validate the window.

Flag (validation)	Description
RDW_NOERASE	Suppresses any pending WM_ERASEBKGND messages.
RDW_NOFRAME	Suppresses any pending WM_NCPAINT messages. This flag must be used with RDW_VALIDATE and is typically used with RDW_NOCHILDREN. RDW_NOFRAME should be used with care, as it could cause parts of a window to be painted improperly.
RDW_NOINTERNALPAINT	Suppresses any pending internal WM_PAINT messages. This flag does not affect WM_PAINT messages resulting from a non-NULL update area.
RDW_VALIDATE	Validates lprcUpdate or hrgnUpdate (only one may be non-NULL). If both are NULL, the entire window is validated. This flag does not affect internal WM_PAINT messages.

The following flags control when repainting occurs. RedrawWindow will not repaint unless one of these flags is specified.

Flag	Description
RDW_ERASENOW	Causes the affected windows (as specified by the RDW_ALLCHILDREN and RDW_NOCHILDREN flags) to receive WM_NCPAINT and WM_ERASEBKGND messages, if necessary, before the function returns. WM_PAINT messages are received at the ordinary time.
RDW_UPDATENOW	Causes the affected windows (as specified by the RDW_ALLCHILDREN and RDW_NOCHILDREN flags) to receive WM_NCPAINT, WM_ERASEBKGND, and WM_PAINT messages, if necessary, before the function returns.

By default, the windows affected by RedrawWindow depend on whether the specified window has the WS_CLIPCHILDREN style. Child windows that are not the WS_CLIPCHILDREN style are unaffected; non-WS_CLIPCHILDREN windows are recursively validated or invalidated until a WS_CLIPCHILDREN window is encountered. The following flags control which windows are affected by the RedrawWindow function.

Flag	Description
RDW_ALLCHILDREN	Includes child windows, if any, in the repainting operation.
RDW_NOCHILDREN	Excludes child windows, if any, from the repainting operation.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Windows NT/ 2000: To get extended error information, call GetLastError.

Remarks

When RedrawWindow is used to invalidate part of the desktop window, the desktop window does not receive a WM_PAINT message. To repaint the desktop, an application uses the RDW_ERASE flag to generate a WM_ERASEBKGND message.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.335 RegisterClass

The RegisterClass function registers a window class for subsequent use in calls to the CreateWindow or CreateWindowEx function.

Note The RegisterClass function has been superseded by the RegisterClassEx function. You can still use RegisterClass, however, if you do not need to set the class small icon.

```
RegisterClass: procedure
(
    var lpWndClass  :WNDCLASS
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__RegisterClassA@4" );
```

Parameters

lpWndClass

[in] Pointer to a WNDCLASS structure. You must fill the structure with the appropriate class attributes before passing it to the function.

Return Values

If the function succeeds, the return value is a class atom that uniquely identifies the class being registered. This atom can only be used by the CreateWindow, CreateWindowEx, GetClassInfo, GetClassInfoEx, FindWindow, FindWindowEx, and UnregisterClass functions and the IActiveIMMApp::FilterClientWindows method.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

If you register the window class by using RegisterClassA, the application tells the system that the windows of the created class expect messages with text or character parameters to use the ANSI character set; if you register it by using RegisterClassW, the application requests that the system pass text parameters of messages as Unicode. The IsWindowUnicode function enables applications to query the nature of each window. For more information on ANSI and Unicode functions in the Win32 API, see Function Prototypes.

All window classes that an application registers are unregistered when it terminates.

Windows 95: All window classes registered by a .dll are unregistered when the .dll is unloaded.

Windows NT/2000: No window classes registered by a .dll are unregistered when the .dll is unloaded. A .dll must explicitly unregister its classes when it is unloaded.

Windows 95: RegisterClass fails if the cbWndExtra or cbClsExtra member of the WNDCLASS structure contains more than 40 bytes.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.336 RegisterClassEx

The RegisterClassEx function registers a window class for subsequent use in calls to the CreateWindow or CreateWindowEx function.

```
RegisterClassEx: procedure
```

```
(
    var lpwcx       :WNDCLASSEX
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__RegisterClassExA@4" );
```

Parameters

lpwcx

[in] Pointer to a WNDCLASSEX structure. You must fill the structure with the appropriate class attributes before passing it to the function.

Return Values

If the function succeeds, the return value is a class atom that uniquely identifies the class being registered. This atom can only be used by the CreateWindow, CreateWindowEx, GetClassInfo, GetClassInfoEx, FindWindow, FindWindowEx, and UnregisterClass functions and the IActiveIMMApp::FilterClientWindows method.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

If you register the window class by using RegisterClassExA, the application tells the system that the windows of the created class expect messages with text or character parameters to use the ANSI character set; if you register it by using RegisterClassExW, the application requests that the system pass text parameters of messages as Unicode. The IsWindowUnicode function enables applications to query the nature of each window. For more information on ANSI and Unicode functions in the Win32 API, see Function Prototypes.

All window classes that an application registers are unregistered when it terminates.

Windows 95: All window classes registered by a .dll are unregistered when the .dll is unloaded.

Windows NT/2000: No window classes registered by a .dll are unregistered when the .dll is unloaded. A .dll must explicitly unregister its classes when it is unloaded.

Windows 95: RegisterClassEx fails if the cbWndExtra or cbClsExtra member of the WNDCLASSEX structure contains more than 40 bytes.

Requirements

Windows NT/2000: Requires Windows NT 4.0 or later.

Windows 95/98: Requires Windows 95 or later.

3.337 RegisterClipboardFormat

The RegisterClipboardFormat function registers a new clipboard format. This format can then be used as a valid clipboard format.

```
RegisterClipboardFormat: procedure
(
    lpszFormat      :string
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__RegisterClipboardFormatA@4" );
```

Parameters

lpszFormat

[in] Pointer to a null-terminated string that names the new format.

Return Values

If the function succeeds, the return value identifies the registered clipboard format.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

If a registered format with the specified name already exists, a new format is not registered and the return value identifies the existing format. This enables more than one application to copy and paste data using the same registered clipboard format. Note that the format name comparison is case-insensitive.

Registered clipboard formats are identified by values in the range 0xC000 through 0xFFFF.

When registered clipboard formats are placed on or retrieved from the clipboard, they must be in the form of an HGLOBAL value.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.338 RegisterDeviceNotification

The RegisterDeviceNotification function specifies the device or type of device for which a window will receive notifications.

```
RegisterDeviceNotification: procedure
(
        hRecipient          :dword;
    var NotificationFilter  :var;
        Flags               :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__RegisterDeviceNotificationA@12" );
```

Parameters

hRecipient

[in] Handle to the window that will receive device events for the devices specified in the NotificationFilter parameter. The same window handle can be used in multiple calls to RegisterDeviceNotification.

Services can specify either a window handle or service status handle.

NotificationFilter

[in] Pointer to a block of data that specifies the type of device for which notifications should be sent. This block always begins with the DEV_BROADCAST_HDR structure. The data following this header is dependent on the value of the dbch_devicetype member.

Flags

[in] Specifies the handle type. This parameter can be one of the following values.

Type	Description
DEVICE_NOTIFY_WINDOW_HANDLE	The hRecipient parameter is a window handle.

DEVICE_NOTIFY_SERVICE_HAN The hRecipient parameter is a service status handle.
DLE

Return Values

If the function succeeds, the return value is a device notification handle.

If the function fails, the return value is NULL. To get extended error information, call GetLastError.

Remarks

Applications send event notifications using the BroadcastSystemMessage function. Any application with a top-level window can receive basic notifications by processing the WM_DEVICECHANGE message. Applications can use the RegisterDeviceNotification function to register to receive device notifications.

Services can use the RegisterDeviceNotification function to register to receive device notifications. If a service specifies a window handle in the hRecipient parameter, the notifications are sent to the window procedure. If hRecipient is a service status handle, the notifications are sent to the service control handler. For more information about the service control handler, see HandlerEx.

Device notification handles returned by RegisterDeviceNotification must be closed by calling the UnregisterDeviceNotification function when they are no longer needed.

Requirements

Windows NT/2000: Requires Windows 2000 or later.

Windows 95/98: Requires Windows 98 or later.

3.339 RegisterHotKey

The RegisterHotKey function defines a system-wide hot key.

```
RegisterHotKey: procedure
(
    hWnd            :dword;
    id              :dword;
    fsModifiers     :dword;
    vk              :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__RegisterHotKey@16" );
```

Parameters

hWnd

[in] Handle to the window that will receive WM_HOTKEY messages generated by the hot key. If this parameter is NULL, WM_HOTKEY messages are posted to the message queue of the calling thread and must be processed in the message loop.

id

[in] Specifies the identifier of the hot key. No other hot key in the calling thread should have the same identifier. An application must specify a value in the range 0x0000 through 0xBFFF. A shared dynamic-link library (DLL) must specify a value in the range 0xC000 through 0xFFFF (the range returned by the GlobalAddAtom function). To avoid conflicts with hot-key identifiers defined by other shared DLLs, a DLL should use the GlobalAddAtom function to obtain the hot-key identifier.

fsModifiers

[in] Specifies keys that must be pressed in combination with the key specified by the nVirtKey parameter in

order to generate the WM_HOTKEY message. The fsModifiers parameter can be a combination of the following values.

Value	Meaning
MOD_ALT	Either ALT key must be held down.
MOD_CONTROL	Either CTRL key must be held down.
MOD_SHIFT	Either SHIFT key must be held down.
MOD_WIN	Either WINDOWS key was held down. These keys are labeled with the Microsoft Windows logo.

vk

[in] Specifies the virtual-key code of the hot key.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

When a key is pressed, the system looks for a match against all hot keys. Upon finding a match, the system posts the WM_HOTKEY message to the message queue of the thread that registered the hot key. This message is posted to the beginning of the queue so it is removed by the next iteration of the message loop.

This function cannot associate a hot key with a window created by another thread.

RegisterHotKey fails if the keystrokes specified for the hot key have already been registered by another hot key.

If the window identified by the hWnd parameter already registered a hot key with the same identifier as that specified by the id parameter, the new values for the fsModifiers and vk parameters replace the previously specified values for these parameters.

Windows NT4 and Windows 2000: The F12 key is reserved for use by the debugger at all times, so it should not be registered as a hot key. Even when you are not debugging an application, F12 is reserved in case a kernel-mode debugger or a just-in-time debugger is resident.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.340 RegisterWindowMessage

The RegisterWindowMessage function defines a new window message that is guaranteed to be unique throughout the system. The message value can be used when sending or posting messages.

```
RegisterWindowMessage: procedure
(
    lpString        :string
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__RegisterWindowMessageA@4" );
```

Parameters

lpString

[in] Pointer to a null-terminated string that specifies the message to be registered.

Return Values

If the message is successfully registered, the return value is a message identifier in the range 0xC000 through 0xFFFF.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

The RegisterWindowMessage function is typically used to register messages for communicating between two cooperating applications.

If two different applications register the same message string, the applications return the same message value. The message remains registered until the session ends.

Only use RegisterWindowMessage when more than one application must process the same message. For sending private messages within a window class, an application can use any integer in the range WM_USER through 0x7FFF. (Messages in this range are private to a window class, not to an application. For example, predefined control classes such as BUTTON, EDIT, LISTBOX, and COMBOBOX may use values in this range.)

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.341 ReleaseCapture

The ReleaseCapture function releases the mouse capture from a window in the current thread and restores normal mouse input processing. A window that has captured the mouse receives all mouse input, regardless of the position of the cursor, except when a mouse button is clicked while the cursor hot spot is in the window of another thread.

```
ReleaseCapture: procedure;
    @stdcall;
    @returns( "eax" );
    @external( "__imp__ReleaseCapture@0" );
```

Parameters

This function has no parameters.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

An application calls this function after calling the SetCapture function.

Windows 95: Calling ReleaseCapture causes the window that is losing the mouse capture to receive a WM_CAPTURECHANGED message.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.342 ReleaseDC

The ReleaseDC function releases a device context (DC), freeing it for use by other applications. The effect of the ReleaseDC function depends on the type of DC. It frees only common and window DCs. It has no effect on class or private DCs.

```
ReleaseDC: procedure
(
    hWnd            :dword;
    hDC             :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__ReleaseDC@8" );
```

Parameters

hWnd

[in] Handle to the window whose DC is to be released.

hDC

[in] Handle to the DC to be released.

Return Values

The return value indicates whether the DC was released. If the DC was released, the return value is 1.

If the DC was not released, the return value is zero.

Remarks

The application must call the ReleaseDC function for each call to the GetWindowDC function and for each call to the GetDC function that retrieves a common DC.

An application cannot use the ReleaseDC function to release a DC that was created by calling the CreateDC function; instead, it must use the DeleteDC function. ReleaseDC must be called from the same thread that called GetDC.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.343 RemoveMenu

The RemoveMenu function deletes a menu item or detaches a submenu from the specified menu. If the menu item opens a drop-down menu or submenu, RemoveMenu does not destroy the menu or its handle, allowing the menu to be reused. Before this function is called, the GetSubMenu function should retrieve a handle to the drop-down menu or submenu.

```
RemoveMenu: procedure
(
    hMenu           :dword;
    uPosition       :dword;
    uFlags          :dword
);
    @stdcall;
    @returns( "eax" );
```

```
    @external( "__imp__RemoveMenu@12" );
```

Parameters

hMenu

[in] Handle to the menu to be changed.

uPosition

[in] Specifies the menu item to be deleted, as determined by the uFlags parameter.

uFlags

[in] Specifies how the uPosition parameter is interpreted. This parameter must be one of the following values.

Value	Meaning
MF_BYCOMMAND	Indicates that uPosition gives the identifier of the menu item. If neither the MF_BYCOMMAND nor MF_BYPOSITION flag is specified, the MF_BYCOMMAND flag is the default flag.
MF_BYPOSITION	Indicates that uPosition gives the zero-based relative position of the menu item.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

The application must call the DrawMenuBar function whenever a menu changes, whether or not the menu is in a displayed window.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.344 RemoveProp

The RemoveProp function removes an entry from the property list of the specified window. The specified character string identifies the entry to be removed.

```
RemoveProp: procedure
(
    hWnd            :dword;
    lpString        :string
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__RemovePropA@8" );
```

Parameters

hWnd

[in] Handle to the window whose property list is to be changed.

lpString

[in] Pointer to a null-terminated character string or contains an atom that identifies a string. If this parameter is an atom, it must have been created using the AddAtom function. The atom, a 16-bit value, must be placed in the

low-order word of lpString; the high-order word must be zero.

Return Values

The return value identifies the specified data. If the data cannot be found in the specified property list, the return value is NULL.

Remarks

The return value is the hData value that was passed to SetProp; it is an application-defined value. Note, this function only destroys the association between the data and the window. If appropriate, the application must free the data handles associated with entries removed from a property list. The application can remove only those properties it has added. It must not remove properties added by other applications or by the system itself.

The RemoveProp function returns the data handle associated with the string so that the application can free the data associated with the handle.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.345 ReplyMessage

The ReplyMessage function is used to reply to a message sent through the SendMessage function without returning control to the function that called SendMessage.

```
ReplyMessage: procedure
(
    lResult         :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__ReplyMessage@4" );
```

Parameters

lResult

[in] Specifies the result of the message processing. The possible values are based on the message sent.

Return Values

If the calling thread was processing a message sent from another thread or process, the return value is nonzero.

If the calling thread was not processing a message sent from another thread or process, the return value is zero.

Remarks

By calling this function, the window procedure that receives the message allows the thread that called SendMessage to continue to run as though the thread receiving the message had returned control. The thread that calls the ReplyMessage function also continues to run.

If the message was not sent through SendMessage or if the message was sent by the same thread, ReplyMessage has no effect.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.346 ReuseDDElParam

The ReuseDDElParam function enables an application to reuse a packed DDE lParam parameter, rather than allocating a new packed lParam. Using this function reduces reallocations for applications that pass packed DDE messages.

```
ReuseDDElParam: procedure
(
    _lParam         :dword;
    msgIn           :dword;
    msgOut          :dword;
    uiLo            :dword;
    uiHi            :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__ReuseDDElParam@20" );
```

Parameters

lParam

[in] Specifies the lParam parameter of the posted DDE message being reused.

msgIn

[in] Specifies the identifier of the received DDE message.

msgOut

[in] Specifies the identifier of the DDE message to be posted. The DDE message will reuse the packed lParam parameter.

uiLo

[in] Specifies the value to be packed into the low-order word of the reused lParam parameter.

uiHi

[in] Specifies the value to be packed into the high-order word of the reused lParam parameter.

Return Values

The return value is the new lParam value.

Remarks

The return value must be posted as the lParam parameter of a DDE message; it must not be used for any other purpose. Once the return value is posted, the posting application need not perform any action to dispose of the lParam parameter.

Use ReuseDDElParam instead of FreeDDElParam if the lParam parameter will be reused in a responding message. ReuseDDElParam returns the lParam appropriate for reuse.

This function allocates or frees lParam parameters as needed, depending on the packing requirements of the incoming and outgoing messages. This reduces reallocations in passing DDE messages.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.347 ScreenToClient

The ScreenToClient function converts the screen coordinates of a specified point on the screen to client coordinates.

```
ScreenToClient: procedure
(
    hWnd            :dword;
    lpPoint         :POINT
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__ScreenToClient@8" );
```

Parameters

hWnd

[in] Handle to the window whose client area will be used for the conversion.

lpPoint

[in] Pointer to a POINT structure that specifies the screen coordinates to be converted.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Windows NT/ 2000: To get extended error information, call GetLastError.

Remarks

The function uses the window identified by the hWnd parameter and the screen coordinates given in the POINT structure to compute client coordinates. It then replaces the screen coordinates with the client coordinates. The new coordinates are relative to the upper-left corner of the specified window's client area.

The ScreenToClient function assumes the specified point is in screen coordinates.

All coordinates are in device units.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.348 ScrollDC

The ScrollDC function scrolls a rectangle of bits horizontally and vertically.

```
ScrollDC: procedure
(
        hDC         :dword;
        dx          :dword;
        dy          :dword;
    var lprcScroll  :RECT;
    var lprcClip    :RECT;
        hrgnUpdate  :dword;
    var lprcUpdate  :RECT
);
```

```
@stdcall;
@returns( "eax" );
@external( "__imp__ScrollDC@28" );
```

Parameters

hDC

[in] Handle to the device context that contains the bits to be scrolled.

dx

[in] Specifies the amount, in device units, of horizontal scrolling. This parameter must be a negative value to scroll to the left.

dy

[in] Specifies the amount, in device units, of vertical scrolling. This parameter must be a negative value to scroll up.

lprcScroll

[in] Pointer to a RECT structure containing the coordinates of the bits to be scrolled. The only bits affected by the scroll operation are bits in the intersection of this rectangle and the rectangle specified by lprcClip. If lprcScroll is NULL, the entire client area is used.

lprcClip

[in] Pointer to a RECT structure containing the coordinates of the clipping rectangle. The only bits that will be painted are the bits that remain inside this rectangle after the scroll operation has been completed. If lprc-Clip is NULL, the entire client area is used.

hrgnUpdate

[in] Handle to the region uncovered by the scrolling process. ScrollDC defines this region; it is not necessarily a rectangle.

lprcUpdate

[out] Pointer to a RECT structure that receives the coordinates of the rectangle bounding the scrolling update region. This is the largest rectangular area that requires repainting. When the function returns, the values in the structure are in client coordinates, regardless of the mapping mode for the specified device context. This allows applications to use the update region in a call to the InvalidateRgn function, if required.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

If the lprcUpdate parameter is NULL, the system does not compute the update rectangle. If both the hrgnUpdate and lprcUpdate parameters are NULL, the system does not compute the update region. If hrgnUpdate is not NULL, the system proceeds as though it contains a valid handle to the region uncovered by the scrolling process (defined by ScrollDC).

When you must scroll the entire client area of a window, use the ScrollWindowEx function.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.349 ScrollWindow

The ScrollWindow function scrolls the contents of the specified window's client area.

Note The ScrollWindow function is provided for backward compatibility. New applications should use the ScrollWindowEx function.

```
ScrollWindow: procedure
(
        hWnd        :dword;
        XAmount     :dword;
        YAmount     :dword;
    var lpRect      :RECT;
    var lpClipRect  :RECT
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__ScrollWindow@20" );
```

Parameters

hWnd

[in] Handle to the window where the client area is to be scrolled.

XAmount

[in] Specifies the amount, in device units, of horizontal scrolling. If the window being scrolled has the CS_OWNDC or CS_CLASSDC style, then this parameter uses logical units rather than device units. This parameter must be a negative value to scroll the content of the window to the left.

YAmount

[in] Specifies the amount, in device units, of vertical scrolling. If the window being scrolled has the CS_OWNDC or CS_CLASSDC style, then this parameter uses logical units rather than device units. This parameter must be a negative value to scroll the content of the window up.

lpRect

[in] Pointer to the RECT structure specifying the portion of the client area to be scrolled. If this parameter is NULL, the entire client area is scrolled.

lpClipRect

[in] Pointer to the RECT structure containing the coordinates of the clipping rectangle. Only device bits within the clipping rectangle are affected. Bits scrolled from the outside of the rectangle to the inside are painted; bits scrolled from the inside of the rectangle to the outside are not painted.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

If the caret is in the window being scrolled, ScrollWindow automatically hides the caret to prevent it from being erased and then restores the caret after the scrolling is finished. The caret position is adjusted accordingly.

The area uncovered by ScrollWindow is not repainted, but it is combined into the window's update region. The application eventually receives a WM_PAINT message notifying it that the region must be repainted. To repaint the uncovered area at the same time the scrolling is in action, call the UpdateWindow function immediately after calling ScrollWindow.

If the lpRect parameter is NULL, the positions of any child windows in the window are offset by the amount specified by the XAmount and YAmount parameters; invalid (unpainted) areas in the window are also offset. ScrollWindow is faster when lpRect is NULL.

If lpRect is not NULL, the positions of child windows are not changed and invalid areas in the window are not offset. To prevent updating problems when lpRect is not NULL, call UpdateWindow to repaint the window before calling ScrollWindow.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.350 ScrollWindowEx

The ScrollWindowEx function scrolls the contents of the specified window's client area.

```
ScrollWindowEx: procedure
(
        hWnd        :dword;
        dx          :dword;
        dy          :dword;
    var prcScroll   :RECT;
    var prcClip     :RECT;
        hrgnUpdate  :dword;
    var prcUpdate   :RECT;
        Flags       :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__ScrollWindowEx@32" );
```

Parameters

hWnd

[in] Handle to the window where the client area is to be scrolled.

dx

[in] Specifies the amount, in device units, of horizontal scrolling. This parameter must be a negative value to scroll to the left.

dy

[in] Specifies the amount, in device units, of vertical scrolling. This parameter must be a negative value to scroll up.

prcScroll

[in] Pointer to a RECT structure that specifies the portion of the client area to be scrolled. If this parameter is NULL, the entire client area is scrolled.

prcClip

[in] Pointer to a RECT structure that contains the coordinates of the clipping rectangle. Only device bits within the clipping rectangle are affected. Bits scrolled from the outside of the rectangle to the inside are painted; bits scrolled from the inside of the rectangle to the outside are not painted. This parameter may be NULL.

hrgnUpdate

[in] Handle to the region that is modified to hold the region invalidated by scrolling. This parameter may be NULL.

prcUpdate

[out] Pointer to a RECT structure that receives the boundaries of the rectangle invalidated by scrolling. This parameter may be NULL.

flags

[in] Specifies flags that control scrolling. This parameter can be one of the following values.

Value	Meaning
SW_ERASE	Erases the newly invalidated region by sending a WM_ERASEBKGND message to the window when specified with the SW_INVALIDATE flag.
SW_INVALIDATE	Invalidates the region identified by the hrgnUpdate parameter after scrolling.
SW_SCROLLCHILDREN	Scrolls all child windows that intersect the rectangle pointed to by the prcScroll parameter. The child windows are scrolled by the number of pixels specified by the dx and dy parameters. The system sends a WM_MOVE message to all child windows that intersect the prcScroll rectangle, even if they do not move.
SW_SMOOTHSCROLL	Windows 98, Windows 2000: Scrolls using smooth scrolling. Use the HIWORD portion of the flags parameter to indicate how much time the smooth-scrolling operation should take.

Return Values

If the function succeeds, the return value is SIMPLEREGION (rectangular invalidated region), COMPLEXREGION (nonrectangular invalidated region; overlapping rectangles), or NULLREGION (no invalidated region).

If the function fails, the return value is ERROR. To get extended error information, call GetLastError.

Remarks

If the SW_INVALIDATE and SW_ERASE flags are not specified, ScrollWindowEx does not invalidate the area that is scrolled from. If either of these flags is set, ScrollWindowEx invalidates this area. The area is not updated until the application calls the UpdateWindow function, calls the RedrawWindow function (specifying the RDW_UPDATENOW or RDW_ERASENOW flag), or retrieves the WM_PAINT message from the application queue.

If the window has the WS_CLIPCHILDREN style, the returned areas specified by hrgnUpdate and prcUpdate represent the total area of the scrolled window that must be updated, including any areas in child windows that need updating.

If the SW_SCROLLCHILDREN flag is specified, the system does not properly update the screen if part of a child window is scrolled. The part of the scrolled child window that lies outside the source rectangle is not erased and is not properly redrawn in its new destination. To move child windows that do not lie completely within the rectangle specified by prcScroll, use the DeferWindowPos function. The cursor is repositioned if the SW_SCROLLCHILDREN flag is set and the caret rectangle intersects the scroll rectangle.

All input and output coordinates (for prcScroll, prcClip, prcUpdate, and hrgnUpdate) are determined as client coordinates, regardless of whether the window has the CS_OWNDC or CS_CLASSDC class style. Use the LPtoDP and DPtoLP functions to convert to and from logical coordinates, if necessary.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.351 SendDlgItemMessage

The SendDlgItemMessage function sends a message to the specified control in a dialog box.

```
SendDlgItemMessage: procedure
(
    hDlg            :dword;
    nIDLdgItem      :dword;
    _Msg            :dword;
    _wParam         :dword;
    _lParam         :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__SendDlgItemMessageA@20" );
```

Parameters

hDlg

[in] Handle to the dialog box that contains the control.

nIDDlgItem

[in] Specifies the identifier of the control that receives the message.

Msg

[in] Specifies the message to be sent.

wParam

[in] Specifies additional message-specific information.

lParam

[in] Specifies additional message-specific information.

Return Values

The return value specifies the result of the message processing and depends on the message sent.

Remarks

The SendDlgItemMessage function does not return until the message has been processed.

Using SendDlgItemMessage is identical to retrieving a handle to the specified control and calling the SendMessage function.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.352 SendInput

The SendInput function synthesizes keystrokes, mouse motions, and button clicks.

```
SendInput: procedure
(
        nInputs     :dword;
```

```
    var pInputs     :INPUT;
        cbSize      :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__SendInput@12" );
```

Parameters

nInputs

[in] Specifies the number of structures in the pInputs array.

pInputs

[in] Pointer to an array of INPUT structures. Each structure represents an event to be inserted into the keyboard or mouse input stream.

cbSize

[in] Specifies the size, in bytes, of an INPUT structure. If cbSize is not the size of an INPUT structure, the function will fail.

Return Values

The function returns the number of events that it successfully inserted into the keyboard or mouse input stream. If the function returns zero, the input was already blocked by another thread.

To get extended error information, call GetLastError.

Remarks

The SendInput function inserts the events in the INPUT structures serially into the keyboard or mouse input stream. These events aren't interspersed with other keyboard or mouse input events inserted either by the user (with the keyboard or mouse) or by calls to keybd_event, mouse_event, or other calls to SendInput.

This function does not reset the keyboard's current state. Any keys that are already pressed when the function is called might interfere with the events that this function generates. To avoid this problem, check the keyboard's state with the GetAsyncKeyState function and correct as necessary.

Requirements

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.

Windows 95/98: Requires Windows 98 or later.

3.353 SendMessage

The SendMessage function sends the specified message to a window or windows. It calls the window procedure for the specified window and does not return until the window procedure has processed the message.

To send a message and return immediately, use the SendMessageCallback or SendNotifyMessage function. To post a message to a thread's message queue and return immediately, use the PostMessage or PostThreadMessage function.

```
SendMessage: procedure
(
    hWnd            :dword;
    _Msg            :dword;
    _wParam         :dword;
    _lParam         :dword
);
```

```
    @stdcall;
    @returns( "eax" );
    @external( "__imp__SendMessageA@16" );
```

Parameters

hWnd

[in] Handle to the window whose window procedure will receive the message. If this parameter is HWND_BROADCAST, the message is sent to all top-level windows in the system, including disabled or invisible unowned windows, overlapped windows, and pop-up windows; but the message is not sent to child windows.

Msg

[in] Specifies the message to be sent.

wParam

[in] Specifies additional message-specific information.

lParam

[in] Specifies additional message-specific information.

Return Values

The return value specifies the result of the message processing; it depends on the message sent.

Remarks

Applications that need to communicate using HWND_BROADCAST should use the RegisterWindowMessage function to obtain a unique message for inter-application communication.

If the specified window was created by the calling thread, the window procedure is called immediately as a subroutine. If the specified window was created by a different thread, the system switches to that thread and calls the appropriate window procedure. Messages sent between threads are processed only when the receiving thread executes message retrieval code. The sending thread is blocked until the receiving thread processes the message. However, the sending thread will process messages from its queue while waiting for its message to be processed. To prevent this, use SendMessageTimeout with SMTO_BLOCK set.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.354 SendMessageCallback

The SendMessageCallback function sends the specified message to a window or windows. It calls the window procedure for the specified window and returns immediately. After the window procedure processes the message, the system calls the specified callback function, passing the result of the message processing and an application-defined value to the callback function.

```
SendMessageCallback: procedure
(
    hWnd            :dword;
    _Msg            :dword;
    _wParam         :dword;
    _lParam         :dword;
    lpCallBack      :SENDASYNCPROC;
    dwData          :dword
);
```

```
@stdcall;
@returns( "eax" );
@external( "__imp__SendMessageCallbackA@24" );
```

Parameters

hWnd

[in] Handle to the window whose window procedure will receive the message. If this parameter is HWND_BROADCAST, the message is sent to all top-level windows in the system, including disabled or invisible unowned windows, overlapped windows, and pop-up windows; but the message is not sent to child windows.

Msg

[in] Specifies the message to be sent.

wParam

[in] Specifies additional message-specific information.

lParam

[in] Specifies additional message-specific information.

lpCallBack

[in] Pointer to a callback function that the system calls after the window procedure processes the message. For more information, see SendAsyncProc.

If hWnd is HWND_BROADCAST, the system calls the SendAsyncProc callback function once for each top-level window.

dwData

[in] Specifies an application-defined value to be sent to the callback function pointed to by the lpCallBack parameter.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

If you send a message in the range below WM_USER to the asynchronous message functions (PostMessage, SendNotifyMessage, and SendMessageCallback), its message parameters can not include pointers. Otherwise, the operation will fail. The functions will return before the receiving thread has had a chance to process the message and the sender will free the memory before it is used.

Applications that need to communicate using HWND_BROADCAST should use the RegisterWindowMessage function to obtain a unique message for inter-application communication.

The callback function is called only when the thread that called SendMessageCallback also calls GetMessage, PeekMessage, or WaitMessage.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.355 SendMessageTimeout

The SendMessageTimeout function sends the specified message to a window or windows. The function calls the window procedure for the specified window and, if the specified window belongs to a different thread, does not return until the window procedure has processed the message or the specified time-out period has elapsed. If the window receiving the message belongs to the same queue as the current thread, the window procedure is called directly — the time-out value is ignored.

```
SendMessageTimeout: procedure
(
        hWnd        :dword;
        _Msg        :dword;
        _wParam     :dword;
        _lParam     :dword;
        fuFlags     :dword;
        uTimeout    :dword;
    var lpdwResult  :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__SendMessageTimeoutA@28" );
```

Parameters

hWnd

[in] Handle to the window whose window procedure will receive the message.

If this parameter is HWND_BROADCAST, the message is sent to all top-level windows in the system, including disabled or invisible unowned windows. The function does not return until each window has timed out. Therefore, the total wait time can be up to uTimeout times the number of top-level windows.

Msg

[in] Specifies the message to be sent.

wParam

[in] Specifies additional message-specific information.

lParam

[in] Specifies additional message-specific information.

fuFlags

[in] Specifies how to send the message. This parameter can be one or more of the following values.

Value	Meaning
SMTO_ABORTIFHUNG	Returns without waiting for the time-out period to elapse if the receiving process appears to be in a "hung" state.
SMTO_BLOCK	Prevents the calling thread from processing any other requests until the function returns.
SMTO_NORMAL	The calling thread is not prevented from processing other requests while waiting for the function to return.
SMTO_NOTIMEOUTIFNOTHUNG	Windows 2000: Does not return when the time-out period elapses if the receiving thread is not hung.

uTimeout

[in] Specifies the duration, in milliseconds, of the time-out period. If the message is a broadcast message, each window can us the full time-out period. For example, if you specify a 5 second time-out period and there are three top-level windows that fail to process the message, you could have up to a 15 second delay.

lpdwResult

[in] Receives the result of the message processing. This value depends on the message sent.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails or time out, the return value is zero. To get extended error information, call GetLastError. If GetLastError returns zero, then the function timed out. SendMessageTimeout does not provide information about individual windows timing out if HWND_BROADCAST is used.

Remarks

This function considers a thread to be hung if it has not called GetMessage or a similar function within 5 seconds.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.356 SendNotifyMessage

The SendNotifyMessage function sends the specified message to a window or windows. If the window was created by the calling thread, SendNotifyMessage calls the window procedure for the window and does not return until the window procedure has processed the message. If the window was created by a different thread, SendNotifyMessage passes the message to the window procedure and returns immediately; it does not wait for the window procedure to finish processing the message.

```
SendNotifyMessage: procedure
(
    hWnd            :dword;
    _Msg            :dword;
    _wParam         :dword;
    _lParam         :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__SendNotifyMessageA@16" );
```

Parameters

hWnd

[in] Handle to the window whose window procedure will receive the message. If this parameter is HWND_BROADCAST, the message is sent to all top-level windows in the system, including disabled or invisible unowned windows, overlapped windows, and pop-up windows; but the message is not sent to child windows.

Msg

[in] Specifies the message to be sent.

wParam

[in] Specifies additional message-specific information.

lParam

[in] Specifies additional message-specific information.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

If you send a message in the range below WM_USER to the asynchronous message functions (PostMessage, SendNotifyMessage, and SendMessageCallback), its message parameters can not include pointers. Otherwise, the operation will fail. The functions will return before the receiving thread has had a chance to process the message and the sender will free the memory before it is used.

Applications that need to communicate using HWND_BROADCAST should use the RegisterWindowMessage function to obtain a unique message for inter-application communication.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.357 SetActiveWindow

The SetActiveWindow function activates a window. The window must be attached to the calling thread's message queue.

```
SetActiveWindow: procedure
(
    hWnd            :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__SetActiveWindow@4" );
```

Parameters

hWnd

[in] Handle to the top-level window to be activated.

Return Values

If the function succeeds, the return value is the handle to the window that was previously active.

If the function fails, the return value is NULL. To get extended error information, call GetLastError.

Remarks

The SetActiveWindow function activates a window, but not if the application is in the background. The window will be brought into the foreground (top of Z order) if its application is in the foreground when the system activates the window.

If the window identified by the hWnd parameter was created by the calling thread, the active window status of the calling thread is set to hWnd. Otherwise, the active window status of the calling thread is set to NULL.

By using the AttachThreadInput function, a thread can attach its input processing to another thread. This allows a thread to call SetActiveWindow to activate a window attached to another thread's message queue.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.358 SetCapture

The SetCapture function sets the mouse capture to the specified window belonging to the current thread. SetCapture captures mouse input either when the mouse is over the capturing window, or when the mouse button was pressed while the mouse was over the capturing window and the button is still down. Only one window at a time can capture the mouse.

If the mouse cursor is over a window created by another thread, the system will direct mouse input to the specified window only if a mouse button is down.

```
SetCapture: procedure
(
    hWnd            :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__SetCapture@4" );
```

Parameters

hWnd

[in] Handle to the window in the current thread that is to capture the mouse.

Return Values

The return value is a handle to the window that had previously captured the mouse. If there is no such window, the return value is NULL.

Remarks

Only the foreground window can capture the mouse. When a background window attempts to do so, the window receives messages only for mouse events that occur when the cursor hot spot is within the visible portion of the window. Also, even if the foreground window has captured the mouse, the user can still click another window, bringing it to the foreground.

When the window no longer requires all mouse input, the thread that created the window should call the ReleaseCapture function to release the mouse.

This function cannot be used to capture mouse input meant for another process.

When the mouse is captured, menu hotkeys and other keyboard accelerators do not work.

Windows 95: Calling SetCapture causes the window that is losing the mouse capture to receive a WM_CAPTURECHANGED message.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.359 SetCaretBlinkTime

The SetCaretBlinkTime function sets the caret blink time to the specified number of milliseconds. The blink time is the elapsed time, in milliseconds, required to invert the caret's pixels.

```
SetCaretBlinkTime: procedure
(
    uMSeconds       :dword
);
```

```
    @stdcall;
    @returns( "eax" );
    @external( "__imp__SetCaretBlinkTime@4" );
```

Parameters

uMSeconds

[in] Specifies the new blink time, in milliseconds.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

The user can set the blink time using the Control Panel. Applications should respect the setting that the user has chosen. The SetCaretBlinkTime function should only be used by application that allow the user to set the blink time, such as a Control Panel applet.

If you change the blink time, subsequently activated applications will use the modified blink time, even if you restore the previous blink time when you lose the keyboard focus or become inactive. This is due to the multi-threaded environment, where deactivation of your application is not synchronized with the activation of another application. This feature allows the system to activate another application even if the current application is hung.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.360 SetCaretPos

The SetCaretPos function moves the caret to the specified coordinates. If the window that owns the caret was created with the CS_OWNDC class style, then the specified coordinates are subject to the mapping mode of the device context associated with that window.

```
SetCaretPos: procedure
(
    X           :dword;
    Y           :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__SetCaretPos@8" );
```

Parameters

X

[in] Specifies the new x-coordinate of the caret.

Y

[in] Specifies the new y-coordinate of the caret.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

SetCaretPos moves the caret whether or not the caret is hidden.

The system provides one caret per queue. A window should create a caret only when it has the keyboard focus or is active. The window should destroy the caret before losing the keyboard focus or becoming inactive. A window can set the caret position only if it owns the caret.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.361 SetClassLong

The SetClassLong function replaces the specified 32-bit (long) value at the specified offset into the extra class memory or the WNDCLASSEX structure for the class to which the specified window belongs.

Note This function has been superseded by the SetClassLongPtr function. To write code that is compatible with both 32-bit and 64-bit versions of Windows, use SetClassLongPtr.

```
SetClassLong: procedure
(
    hWnd            :dword;
    nIndex          :dword;
    dwNewLong       :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__SetClassLongA@12" );
```

Parameters

hWnd

[in] Handle to the window and, indirectly, the class to which the window belongs.

nIndex

[in] Specifies the 32-bit value to replace. To set a 32-bit value in the extra class memory, specify the positive, zero-based byte offset of the value to be set. Valid values are in the range zero through the number of bytes of extra class memory, minus four; for example, if you specified 12 or more bytes of extra class memory, a value of 8 would be an index to the third 32-bit integer. To set any other value from the WNDCLASSEX structure, specify one of the following values.

Value	Action
GCL_CBCLSEXTRA	Sets the size, in bytes, of the extra memory associated with the class. Setting this value does not change the number of extra bytes already allocated.
GCL_CBWNDEXTRA	Sets the size, in bytes, of the extra window memory associated with each window in the class. Setting this value does not change the number of extra bytes already allocated. For information on how to access this memory, see SetWindowLong.
GCL_HBRBACKGROUND	Replaces a handle to the background brush associated with the class.
GCL_HCURSOR	Replaces a handle to the cursor associated with the class.
GCL_HICON	Replaces a handle to the icon associated with the class.

GCL_HICONSM	Replace a handle to the small icon associated with the class.
GCL_HMODULE	Replaces a handle to the module that registered the class.
GCL_MENUNAME	Replaces the address of the menu name string. The string identifies the menu resource associated with the class.
GCL_STYLE	Replaces the window-class style bits.
GCL_WNDPROC	Replaces the address of the window procedure associated with the class.

dwNewLong

[in] Specifies the replacement value.

Return Values

If the function succeeds, the return value is the previous value of the specified 32-bit integer. If the value was not previously set, the return value is zero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

If you use the SetClassLong function and the GCL_WNDPROC index to replace the window procedure, the window procedure must conform to the guidelines specified in the description of the WindowProc callback function.

Calling SetClassLong with the GCL_WNDPROC index creates a subclass of the window class that affects all windows subsequently created with the class. An application can subclass a system class, but should not subclass a window class created by another process.

Reserve extra class memory by specifying a nonzero value in the cbClsExtra member of the WNDCLASSEX structure used with the RegisterClassEx function.

Use the SetClassLong function with care. For example, it is possible to change the background color for a class by using SetClassLong, but this change does not immediately repaint all windows belonging to the class.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.362 SetClassWord

The SetClassWord function replaces the 16-bit (word) value at the specified offset into the extra class memory for the window class to which the specified window belongs.

Note This function is provided only for compatibility with 16-bit versions of Windows. Win32-based applications should use the SetClassLong function.

```
SetClassWord: procedure
(
    hWnd            :dword;
    nIndex          :dword;
    wNewWord        :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__SetClassWord@12" );
```

Parameters

hWnd

[in] Handle to the window and, indirectly, the class to which the window belongs.

nIndex

[in] Specifies the zero-based byte offset of the value to be replaced. Valid values are in the range zero through the number of bytes of class memory minus two; for example, if you specified 10 or more bytes of extra class memory, a value of 8 would be an index to the fifth 16-bit integer.

wNewWord

[in] Specifies the replacement value.

Return Values

If the function succeeds, the return value is the previous value of the specified 16-bit integer. If the value was not previously set, the return value is zero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

Reserve extra class memory by specifying a nonzero value in the cbClsExtra member of the WNDCLASS structure used with the RegisterClass function.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.363 SetClipboardData

The SetClipboardData function places data on the clipboard in a specified clipboard format. The window must be the current clipboard owner, and the application must have called the OpenClipboard function. (When responding to the WM_RENDERFORMAT and WM_RENDERALLFORMATS messages, the clipboard owner must not call OpenClipboard before calling SetClipboardData.)

```
SetClipboardData: procedure
(
    uFormat         :dword;
    hMem            :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__SetClipboardData@8" );
```

Parameters

uFormat

[in] Specifies a clipboard format. This parameter can be a registered format or any of the standard clipboard formats. For more information, see Registered Clipboard Formats and Standard Clipboard Formats.

hMem

[in] Handle to the data in the specified format. This parameter can be NULL, indicating that the window provides data in the specified clipboard format (renders the format) upon request. If a window delays rendering, it must process the WM_RENDERFORMAT and WM_RENDERALLFORMATS messages.

After SetClipboardData is called, the system owns the object identified by the hMem parameter. The application can read the data, but must not free the handle or leave it locked until the CloseClipboard function is called. (The application can access the data after calling CloseClipboard). If the hMem parameter identifies a memory object, the object must have been allocated using the GlobalAlloc function with the GMEM_MOVEABLE flag.

Return Values

If the function succeeds, the return value is the handle to the data.

If the function fails, the return value is NULL. To get extended error information, call GetLastError.

Remarks

The uFormat parameter can identify a registered clipboard format, or it can be one of the standard clipboard formats. For more information, see Registered Clipboard Formats and Standard Clipboard Formats.

If an application calls SetClipboardData in response to WM_RENDERFORMAT or WM_RENDERALLFORMATS, the application should not use the handle after SetClipboardData has been called.

The system performs implicit data format conversions between certain clipboard formats when an application calls the GetClipboardData function. For example, if the CF_OEMTEXT format is on the clipboard, a window can retrieve data in the CF_TEXT format. The format on the clipboard is converted to the requested format on demand. For more information, see Synthesized Clipboard Formats.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.364 SetClipboardViewer

The SetClipboardViewer function adds the specified window to the chain of clipboard viewers. Clipboard viewer windows receive a WM_DRAWCLIPBOARD message whenever the content of the clipboard changes.

```
SetClipboardViewer: procedure
(
    hWndNewViewer   :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__SetClipboardViewer@4" );
```

Parameters

hWndNewViewer

[in] Handle to the window to be added to the clipboard chain.

Return Values

If the function succeeds, the return value identifies the next window in the clipboard viewer chain. If an error occurs or there are no other windows in the clipboard viewer chain, the return value is NULL. To get extended error information, call GetLastError.

Remarks

The windows that are part of the clipboard viewer chain, called clipboard viewer windows, must process the clipboard messages WM_CHANGECBCHAIN and WM_DRAWCLIPBOARD. Each clipboard viewer window calls the SendMessage function to pass these messages to the next window in the clipboard viewer chain.

A clipboard viewer window must eventually remove itself from the clipboard viewer chain by calling the ChangeClipboardChain function — for example, in response to the WM_DESTROY message.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.365 SetCursor

The SetCursor function sets the cursor shape.

```
SetCursor: procedure
(
    hCursor         :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__SetCursor@4" );
```

Parameters

hCursor

[in] Handle to the cursor. The cursor must have been created by the CreateCursor function or loaded by the LoadCursor or LoadImage function. If this parameter is NULL, the cursor is removed from the screen.

Windows 95: The width and height of the cursor must be the values returned by the GetSystemMetrics function for SM_CXCURSOR and SM_CYCURSOR. In addition, either the cursor bit depth must match the bit depth of the display or the cursor must be monochrome.

Return Values

The return value is the handle to the previous cursor, if there was one.

If there was no previous cursor, the return value is NULL.

Remarks

The cursor is set only if the new cursor is different from the previous cursor; otherwise, the function returns immediately.

The cursor is a shared resource. A window should set the cursor shape only when the cursor is in its client area or when the window is capturing mouse input. In systems without a mouse, the window should restore the previous cursor before the cursor leaves the client area or before it relinquishes control to another window.

If your application must set the cursor while it is in a window, make sure the class cursor for the specified window's class is set to NULL. If the class cursor is not NULL, the system restores the class cursor each time the mouse is moved.

The cursor is not shown on the screen if the internal cursor display count is less than zero. This occurs if the application uses the ShowCursor function to hide the cursor more times than to show the cursor.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.366 SetCursorPos

The SetCursorPos function moves the cursor to the specified screen coordinates. If the new coordinates are not within the screen rectangle set by the most recent ClipCursor function call, the system automatically adjusts the coordinates so that the cursor stays within the rectangle.

```
SetCursorPos: procedure
(
    X       :dword;
    Y       :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__SetCursorPos@8" );
```

Parameters

X

[in] Specifies the new x-coordinate of the cursor, in screen coordinates.

Y

[in] Specifies the new y-coordinate of the cursor, in screen coordinates.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

The cursor is a shared resource. A window should move the cursor only when the cursor is in its client area.

The calling process must have WINSTA_WRITEATTRIBUTES access to the window station.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.367 SetDebugErrorLevel

The SetDebugErrorLevel function is obsolete. It is provided only for compatibility with 16-bit versions of Windows.

See Also

Debugging Overview, Debugging Functions

3.368 SetDlgItemInt

The SetDlgItemInt function sets the text of a control in a dialog box to the string representation of a specified integer value.

```
SetDlgItemInt: procedure
(
    hDlg        :dword;
    nIDDlgItem  :dword;
    uValue      :dword;
    bSigned     :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__SetDlgItemInt@16" );
```

Parameters

hDlg

[in] Handle to the dialog box that contains the control.

nIDDlgItem

[in] Specifies the control to be changed.

uValue

[in] Specifies the integer value used to generate the item text.

bSigned

[in] Specifies whether the uValue parameter is signed or unsigned. If this parameter is TRUE, uValue is signed. If this parameter is TRUE and uValue is less than zero, a minus sign is placed before the first digit in the string. If this parameter is FALSE, uValue is unsigned.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

To set the new text, this function sends a WM_SETTEXT message to the specified control.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.369 SetDlgItemText

The SetDlgItemText function sets the title or text of a control in a dialog box.

```
SetDlgItemText: procedure
(
    hDlg        :dword;
    nIDDlgItem  :dword;
    lpString    :string
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__SetDlgItemTextA@12" );
```

Parameters

hDlg

[in] Handle to the dialog box that contains the control.

nIDDlgItem

[in] Specifies the control with a title or text to be set.

lpString

[in] Pointer to the null-terminated string that contains the text to be copied to the control.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

The SetDlgItemText function sends a WM_SETTEXT message to the specified control.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.370 SetDoubleClickTime

The SetDoubleClickTime function sets the double-click time for the mouse. A double-click is a series of two clicks of a mouse button, the second occurring within a specified time after the first. The double-click time is the maximum number of milliseconds that may occur between the first and second clicks of a double-click.

```
SetDoubleClickTime: procedure
(
    uInterval   :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__SetDoubleClickTime@4" );
```

Parameters

uInterval

[in] Specifies the number of milliseconds that may occur between the first and second clicks of a double-click. If this parameter is set to zero, the system uses the default double-click time of 500 milliseconds.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

The SetDoubleClickTime function alters the double-click time for all windows in the system.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.371 SetFocus

The SetFocus function sets the keyboard focus to the specified window. The window must be attached to the calling thread's message queue.

```
SetFocus: procedure
(
    hWnd        :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__SetFocus@4" );
```

Parameters

hWnd

[in] Handle to the window that will receive the keyboard input. If this parameter is NULL, keystrokes are ignored.

Return Values

If the function succeeds, the return value is the handle to the window that previously had the keyboard focus. If the hWnd parameter is invalid or the window is not attached to the calling thread's message queue, the return value is NULL. To get extended error information, call GetLastError.

Remarks

The SetFocus function sends a WM_KILLFOCUS message to the window that loses the keyboard focus and a WM_SETFOCUS message to the window that receives the keyboard focus. It also activates either the window that receives the focus or the parent of the window that receives the focus.

If a window is active but does not have the focus, any key pressed will produce the WM_SYSCHAR, WM_SYSKEYDOWN, or WM_SYSKEYUP message. If the VK_MENU key is also pressed, the lParam parameter of the message will have bit 30 set. Otherwise, the messages produced do not have this bit set.

By using the AttachThreadInput function, a thread can attach its input processing to another thread. This allows a thread to call SetFocus to set the keyboard focus to a window attached to another thread's message queue.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.372 SetForegroundWindow

The SetForegroundWindow function puts the thread that created the specified window into the foreground and activates the window. Keyboard input is directed to the window, and various visual cues are changed for the user. The system assigns a slightly higher priority to the thread that created the foreground window than it does to other threads.

```
SetForegroundWindow: procedure
(
    hWnd        :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__SetForegroundWindow@4" );
```

Parameters

hWnd

[in] Handle to the window that should be activated and brought to the foreground.

Return Values

If the window was brought to the foreground, the return value is nonzero.

If the window was not brought to the foreground, the return value is zero.

Remarks

The foreground window is the window at the top of the Z order. It is the window that the user is working with. In a preemptive multitasking environment, you should generally let the user control which window is the foreground window.

Windows 98, Windows 2000: The system restricts which processes can set the foreground window. A process can set the foreground window only if one of the following conditions is true:

- The process is the foreground process.
- The process was started by the foreground process.
- The process received the last input event.
- There is no foreground process.
- The foreground process is being debugged.
- The foreground is not locked (see LockSetForegroundWindow).
- The foreground lock time-out has expired (see SPI_GETFOREGROUNDLOCKTIMEOUT in SystemParametersInfo).
- Windows 2000: No menus are active.

With this change, an application cannot force a window to the foreground while the user is working with another window. Instead, SetForegroundWindow will activate the window (see SetActiveWindow) and call the FlashWindowEx function to notify the user. However, on Windows 98, if a nonforeground thread calls SetForegroundWindow and passes the handle of a window that was not created by the calling thread, the window is not flashed on the taskbar. To have SetForegroundWindow behave the same as it did on Windows 95 and Windows NT 4.0, change the foreground lock timeout value when the application is installed. This can be done from the setup or installation application with the following function call:

```
SystemParametersInfo(SPI_SETFOREGROUNDLOCKTIMEOUT, 0, (LPVOID)0,
SPIF_SENDWININICHANGE | SPIF_UPDATEINIFILE);
```

This method allows SetForegroundWindow on Windows 98 and Windows 2000 to behave the same as Windows 95 and Windows NT 4.0, respectively, for all applications. The setup application should warn the user that this is being done so that the user isn't surprised by the changed behavior. On Windows 2000, the SystemParametersInfo call fails unless the calling thread can change the foreground window, so this must be called from a setup or patch application. For more information, see Foreground and Background Windows.

A process that can set the foreground window can enable another process to set the foreground window by calling the AllowSetForegroundWindow function. The process specified by dwProcessId loses the ability to set the foreground window the next time the user generates input, unless the input is directed at that process, or the next time a process calls AllowSetForegroundWindow, unless that process is specified.

The foreground process can disable calls to SetForegroundWindow by calling the LockSetForegroundWindow function.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.373 SetKeyboardState

The SetKeyboardState function copies a 256-byte array of keyboard key states into the calling thread's keyboard input-state table. This is the same table accessed by the GetKeyboardState and GetKeyState functions. Changes made to this table do not affect keyboard input to any other thread.

```
SetKeyboardState: procedure
(
    var lpKeyState  :var
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__SetKeyboardState@4" );
```

Parameters

lpKeyState

[in] Pointer to a 256-byte array that contains keyboard key states.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

Because the SetKeyboardState function alters the input state of the calling thread and not the global input state of the system, an application cannot use SetKeyboardState to set the NUM LOCK, CAPS LOCK, or SCROLL LOCK (or the Japanese KANA) indicator lights on the keyboard. These can be set or cleared using SendInput to simulate keystrokes.

Windows NT: The keybd_event function can also toggle the NUM LOCK, CAPS LOCK, and SCROLL LOCK keys.

Windows 95: The keybd_event function can toggle only the CAPS LOCK and SCROLL LOCK keys. It cannot toggle the NUM LOCK key.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.374 SetLastErrorEx

The SetLastErrorEx function sets the last-error code.

Currently, this function is identical to the SetLastError function. The second parameter is ignored.

```
SetLastErrorEx: procedure
(
    dwErrCode       :dword;
    dwType          :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__SetLastErrorEx@8" );
```

Parameters

dwErrCode

[in] Specifies the last-error code for the thread.

dwType

This parameter is ignored.

Return Values

This function does not return a value.

Remarks

Error codes are 32-bit values (bit 31 is the most significant bit). Bit 29 is reserved for application-defined error codes; no Win32 API error code has this bit set. If you are defining an error code for your application, set this bit to indicate that the error code has been defined by the application and to ensure that your error code does not conflict with any system-defined error codes.

This function is intended primarily for dynamic-link libraries (DLL). Calling this function after an error occurs allows the DLL to emulate the behavior of the Win32 API.

Most Win32 functions call SetLastError when they fail. Function failure is typically indicated by a return value error code such as zero, NULL, or –1. Some functions call SetLastError under conditions of success; those cases are noted in each function's reference topic.

Applications can retrieve the value saved by this function by using the GetLastError function.

The last-error code is kept in thread local storage so that multiple threads do not overwrite each other's values.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.375 SetMenu

The SetMenu function assigns a new menu to the specified window.

```
SetMenu: procedure
(
    hWnd        :dword;
    hMenu       :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__SetMenu@8" );
```

Parameters

hWnd

[in] Handle to the window to which the menu is to be assigned.

hMenu

[in] Handle to the new menu. If this parameter is NULL, the window's current menu is removed.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

The window is redrawn to reflect the menu change. A menu can be assigned to any window that is not a child window.

The SetMenu function replaces the previous menu, if any, but it does not destroy it. An application should call the DestroyMenu function to accomplish this task.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.376 SetMenuContextHelpId

Associates a Help context identifier with a menu.

```
SetMenuContextHelpId: procedure
(
    hmenu           :dword;
    dwContextHelpID :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__SetMenuContextHelpId@8" );
```

Parameters

hmenu

Handle to the menu with which to associate the Help context identifier.

dwContextHelpId

Help context identifier.

Return Values

Returns nonzero if successful, or zero otherwise.

To get extended error information, call GetLastError.

Remarks

All items in the menu share this identifier. Help context identifiers can't be attached to individual menu items.

3.377 SetMenuDefaultItem

The SetMenuDefaultItem function sets the default menu item for the specified menu.

```
SetMenuDefaultItem: procedure
(
    hMenu       :dword;
    uItem       :dword;
    fByPos      :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__SetMenuDefaultItem@12" );
```

Parameters

hMenu

[in] Handle to the menu to set the default item for.

uItem

[in] Identifier or position of the new default menu item or – 1 for no default item. The meaning of this param-

eter depends on the value of fByPos.

fByPos

[in] Value specifying the meaning of uItem. If this parameter is FALSE, uItem is a menu item identifier. Otherwise, it is a menu item position.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, use the GetLastError function.

Requirements

Windows NT/2000: Requires Windows NT 4.0 or later.

Windows 95/98: Requires Windows 95 or later.

3.378 SetMenuInfo

The SetMenuInfo function sets information for a specified menu.

```
SetMenuInfo: procedure
(
        hmenu   :dword;
    var lpcmi   :CMENUINFO
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__SetMenuInfo@8" );
```

Parameters

hmenu

[in] Handle to a menu.

lpcmi

[in] Pointer to a MENUINFO structure for the menu.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Requirements

Windows NT/2000: Requires Windows 2000 or later.

Windows 95/98: Requires Windows 98 or later.

3.379 SetMenuItemBitmaps

The SetMenuItemBitmaps function associates the specified bitmap with a menu item. Whether the menu item is selected or clear, the system displays the appropriate bitmap next to the menu item.

```
SetMenuItemBitmaps: procedure
(
    hMenu               :dword;
    uPosition           :dword;
```

```
    uFlags              :dword;
    hBitmapUnchecked    :dword;
    hBitmapChecked      :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__SetMenuItemBitmaps@20" );
```

Parameters

hMenu

[in] Handle to the menu containing the item to receive new check-mark bitmaps.

uPosition

[in] Specifies the menu item to be changed, as determined by the uFlags parameter.

uFlags

[in] Specifies how the uPosition parameter is interpreted. The uFlags parameter must be one of the following values.

Value	Meaning
MF_BYCOMMAND	Indicates that uPosition gives the identifier of the menu item. If neither MF_BYCOMMAND nor MF_BYPOSITION is specified, MF_BYCOMMAND is the default flag.
MF_BYPOSITION	Indicates that uPosition gives the zero-based relative position of the menu item.

hBitmapUnchecked

[in] Handle to the bitmap displayed when the menu item is not selected.

hBitmapChecked

[in] Handle to the bitmap displayed when the menu item is selected.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

If either the hBitmapUnchecked or hBitmapChecked parameter is NULL, the system displays nothing next to the menu item for the corresponding check state. If both parameters are NULL, the system displays the default check-mark bitmap when the item is selected, and removes the bitmap when the item is not selected.

When the menu is destroyed, these bitmaps are not destroyed; it is up to the application to destroy them.

The selected and clear bitmaps should be monochrome. The system uses the Boolean AND operator to combine bitmaps with the menu so that the white part becomes transparent and the black part becomes the menu-item color. If you use color bitmaps, the results may be undesirable.

Use the GetSystemMetrics function with the CXMENUCHECK and CYMENUCHECK values to retrieve the bitmap dimensions.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.380 SetMenuItemInfo

The SetMenuItemInfo function changes information about a menu item.

```
SetMenuItemInfo: procedure
(
        hMenu           :dword;
        uItem           :dword;
        fByPosition     :dword;
    var lpmii           :MENUITEMINFO
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__SetMenuItemInfoA@16" );
```

Parameters

hMenu

[in] Handle to the menu that contains the menu item.

uItem

[in] Identifier or position of the menu item to change. The meaning of this parameter depends on the value of fByPosition.

fByPosition

[in] Value specifying the meaning of uItem. If this parameter is FALSE, uItem is a menu item identifier. Otherwise, it is a menu item position.

lpmii

[in] Pointer to a MENUITEMINFO structure that contains information about the menu item and specifies which menu item attributes to change.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, use the GetLastError function.

Remarks

The application must call the DrawMenuBar function whenever a menu changes, whether or not the menu is in a displayed window.

In order for keyboard accelerators to work with bitmap or owner-drawn menu items, the owner of the menu must process the WM_MENUCHAR message. See Owner-Drawn Menus and the WM_MENUCHAR Message for more information.

Requirements

Windows NT/2000: Requires Windows NT 4.0 or later.

Windows 95/98: Requires Windows 95 or later.

3.381 SetMessageExtraInfo

The SetMessageExtraInfo function sets the extra message information for the current thread. Extra message information is an application- or driver-defined value associated with the current thread's message queue. An application can use the GetMessageExtraInfo function to retrieve a thread's extra message information.

```
SetMessageExtraInfo: procedure
(
    _lParam         :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__SetMessageExtraInfo@4" );
```

Parameters

lParam

[in] Specifies the value to associate with the current thread.

Return Values

The return value is the previous value associated with the current thread.

Requirements

Windows NT/2000: Requires Windows NT 3.51 or later.

Windows 95/98: Requires Windows 95 or later.

3.382 SetMessageQueue

The SetMessageQueue function is obsolete. It is provided only for compatibility for 16-bit versions of Windows. It has no meaning in a 32-bit environment, because message queues are expanded dynamically as necessary.

See Also

Messages and Message Queues Overview, Message and Message Queue Functions

---

3.383 SetParent

The SetParent function changes the parent window of the specified child window.

```
SetParent: procedure
(
    hWndChild       :dword;
    hWndNewParent   :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__SetParent@8" );
```

Parameters

hWndChild

[in] Handle to the child window.

hWndNewParent

[in] Handle to the new parent window. If this parameter is NULL, the desktop window becomes the new parent window.

Windows 2000: If this parameter is HWND_MESSAGE, the child window becomes a message-only window.

Return Values

If the function succeeds, the return value is a handle to the previous parent window.

If the function fails, the return value is NULL. To get extended error information, call GetLastError.

Remarks

An application can use the SetParent function to set the parent window of a pop-up, overlapped, or child window. The new parent window and the child window must belong to the same application.

If the window identified by the hWndChild parameter is visible, the system performs the appropriate redrawing and repainting.

For compatibility reasons, SetParent does not modify the WS_CHILD or WS_POPUP window styles of the window whose parent is being changed. Therefore, if hWndNewParent is NULL, you should also clear the WS_CHILD bit and set the WS_POPUP style after calling SetParent. Conversely, if hWndNewParent is not NULL and the window was previously a child of the desktop, you should clear the WS_POPUP style and set the WS_CHILD style before calling SetParent.

Windows 2000: When you change the parent of a window, you should synchronize the UISTATE of both windows. For more information, see WM_CHANGEUISTATE and WM_UPDATEUISTATE.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.384 SetProcessWindowStation

The SetProcessWindowStation function assigns a window station to the calling process. This enables the process to access objects in the window station such as desktops, the clipboard, and global atoms. All subsequent operations on the window station use the access rights granted to hWinSta.

```
SetProcessWindowStation: procedure
(
    hWinSta         :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__SetProcessWindowStation@4" );
```

Parameters

hWinSta

[in] Handle to the window station to be assigned to the calling process. This can be a handle returned by the CreateWindowStation, OpenWindowStation, or GetProcessWindowStation functions.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Requirements

Windows NT/2000: Requires Windows NT 3.51 or later.

Windows 95/98: Unsupported.

3.385 SetProp

The SetProp function adds a new entry or changes an existing entry in the property list of the specified window. The function adds a new entry to the list if the specified character string does not exist already in the list. The new entry contains the string and the handle. Otherwise, the function replaces the string's current handle with the specified handle.

```
SetProp: procedure
(
    hWnd            :dword;
    lpString        :string;
    hData           :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__SetPropA@12" );
```

Parameters

hWnd

[in] Handle to the window whose property list receives the new entry.

lpString

[in] Pointer to a null-terminated string or contains an atom that identifies a string. If this parameter is an atom, it must be a global atom created by a previous call to the GlobalAddAtom function. The atom, a 16-bit value, must be placed in the low-order word of lpString; the high-order word must be zero.

hData

[in] Handle to the data to be copied to the property list. The data handle can identify any value useful to the application.

Return Values

If the data handle and string are added to the property list, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

Before a window is destroyed (that is, before it returns from processing the WM_NCDESTROY message), an application must remove all entries it has added to the property list. The application must use the RemoveProp function to remove the entries.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.386 SetRect

The SetRect function sets the coordinates of the specified rectangle. This is equivalent to assigning the left, top, right, and bottom arguments to the appropriate members of the RECT structure.

```
SetRect: procedure
(
    var lprc        :RECT;
        xLeft       :dword;
```

```
        yTop        :dword;
        xRight      :dword;
        yBottom     :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__SetRect@20" );
```

Parameters

lprc

[out] Pointer to the RECT structure that contains the rectangle to be set.

xLeft

[in] Specifies the x-coordinate of the rectangle's upper-left corner.

yTop

[in] Specifies the y-coordinate of the rectangle's upper-left corner.

xRight

[in] Specifies the x-coordinate of the rectangle's lower-right corner.

yBottom

[in] Specifies the y-coordinate of the rectangle's lower-right corner.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Windows NT/ 2000: To get extended error information, call GetLastError.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.387 SetRectEmpty

The SetRectEmpty function creates an empty rectangle in which all coordinates are set to zero.

```
SetRectEmpty: procedure
(
    var lprc    :RECT
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__SetRectEmpty@4" );
```

Parameters

lprc

[out] Pointer to the RECT structure that contains the coordinates of the rectangle.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Windows NT/ 2000: To get extended error information, call GetLastError.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.388 SetScrollInfo

The SetScrollInfo function sets the parameters of a scroll bar, including the minimum and maximum scrolling positions, the page size, and the position of the scroll box (thumb). The function also redraws the scroll bar, if requested.

```
SetScrollInfo: procedure
(
        hwnd    :dword;
        fnBar   :dword;
    var lpsi    :CSCROLLINFO;
        fRedraw :boolean
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__SetScrollInfo@16" );
```

Parameters

hwnd

[in] Handle to a scroll bar control or a window with a standard scroll bar, depending on the value of the fnBar parameter.

fnBar

[in] Specifies the type of scroll bar for which to set parameters. This parameter can be one of the following values.

Value	Meaning
SB_CTL	Sets the parameters of a scroll bar control. The hwnd parameter must be the handle to the scroll bar control.
SB_HORZ	Sets the parameters of the window's standard horizontal scroll bar.
SB_VERT	Sets the parameters of the window's standard vertical scroll bar.

lpsi

[in] Pointer to a SCROLLINFO structure. Before calling SetScrollInfo, set the cbSize member of the structure to sizeof(SCROLLINFO), set the fMask member to indicate the parameters to set, and specify the new parameter values in the appropriate members.

The fMask member can be one or more of the following values.

Value	Meaning
SIF_DISABLENOSCROLL	Disables the scroll bar instead of removing it, if the scroll bar's new parameters make the scroll bar unnecessary.
SIF_PAGE	Sets the scroll page to the value specified in the nPage member of the SCROLLINFO structure pointed to by lpsi.
SIF_POS	Sets the scroll position to the value specified in the nPos member of the SCROLLINFO structure pointed to by lpsi.

SIF_RANGE	Sets the scroll range to the value specified in the nMin and nMax members of the SCROLLINFO structure pointed to by lpsi.

fRedraw

[in] Specifies whether the scroll bar is redrawn to reflect the changes to the scroll bar. If this parameter is TRUE, the scroll bar is redrawn, otherwise, it is not redrawn.

Return Values

The return value is the current position of the scroll box.

Remarks

The SetScrollInfo function performs range checking on the values specified by the nPage and nPos members of the SCROLLINFO structure. The nPage member must specify a value from 0 to nMax - nMin +1. The nPos member must specify a value between nMin and nMax - max(nPage – 1, 0). If either value is beyond its range, the function sets it to a value that is just within the range.

Requirements

Windows NT/2000: Requires Windows NT 3.51 or later.

Windows 95/98: Requires Windows 95 or later.

3.389 SetScrollPos

The SetScrollPos function sets the position of the scroll box (thumb) in the specified scroll bar and, if requested, redraws the scroll bar to reflect the new position of the scroll box.

Note The SetScrollPos function is provided for backward compatibility. New applications should use the SetScrollInfo function.

```
SetScrollPos: procedure
(
    hWnd        :dword;
    nBar        :dword;
    nPos        :dword;
    bRedraw     :boolean
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__SetScrollPos@16" );
```

Parameters

hWnd

[in] Handle to a scroll bar control or a window with a standard scroll bar, depending on the value of the nBar parameter.

nBar

[in] Specifies the scroll bar to be set. This parameter can be one of the following values.

Value	Meaning
SB_CTL	Sets the position of the scroll box in a scroll bar control. The hWnd parameter must be the handle to the scroll bar control.
SB_HORZ	Sets the position of the scroll box in a window's standard horizontal scroll bar.
SB_VERT	Sets the position of the scroll box in a window's standard vertical scroll bar.

nPos

[in] Specifies the new position of the scroll box. The position must be within the scrolling range. For more information about the scrolling range, see the SetScrollRange function.

bRedraw

[in] Specifies whether the scroll bar is redrawn to reflect the new scroll box position. If this parameter is TRUE, the scroll bar is redrawn. If it is FALSE, the scroll bar is not redrawn.

Return Values

If the function succeeds, the return value is the previous position of the scroll box.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

If the scroll bar is redrawn by a subsequent call to another function, setting the bRedraw parameter to FALSE is useful.

Because the messages that indicate scroll bar position, WM_HSCROLL and WM_VSCROLL, are limited to 16 bits of position data, applications that rely solely on those messages for position data have a practical maximum value of 65,535 for the SetScrollPos function's nPos parameter.

However, because the SetScrollInfo, SetScrollPos, SetScrollRange, GetScrollInfo, GetScrollPos, and GetScrollRange functions support 32-bit scroll bar position data, there is a way to circumvent the 16-bit barrier of the WM_HSCROLL and WM_VSCROLL messages. See GetScrollInfo for a description of the technique.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.390 SetScrollRange

The SetScrollRange function sets the minimum and maximum scroll box positions for the specified scroll bar.

Note The SetScrollRange function is provided for backward compatibility. New applications should use the SetScrollInfo function.

```
SetScrollRange: procedure
(
    hWnd        :dword;
    nBar        :dword;
    nMinPos     :dword;
    nMaxPos     :dword;
    bRedraw     :boolean
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__SetScrollRange@20" );
```

Parameters

hWnd

[in] Handle to a scroll bar control or a window with a standard scroll bar, depending on the value of the nBar parameter.

nBar

[in] Specifies the scroll bar to be set. This parameter can be one of the following values.

Value	Meaning
SB_CTL	Sets the range of a scroll bar control. The hWnd parameter must be the handle to the scroll bar control.
SB_HORZ	Sets the range of a window's standard horizontal scroll bar.
SB_VERT	Sets the range of a window's standard vertical scroll bar.

nMinPos

[in] Specifies the minimum scrolling position.

nMaxPos

[in] Specifies the maximum scrolling position.

bRedraw

[in] Specifies whether the scroll bar should be redrawn to reflect the change. If this parameter is TRUE, the scroll bar is redrawn. If it is FALSE, the scroll bar is not redrawn.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

You can use SetScrollRange to hide the scroll bar by setting nMinPos and nMaxPos to the same value. An application should not call the SetScrollRange function to hide a scroll bar while processing a scroll bar message. New applications should use the ShowScrollBar function to hide the scroll bar.

If the call to SetScrollRange immediately follows a call to the SetScrollPos function, the bRedraw parameter in SetScrollPos must be zero to prevent the scroll bar from being drawn twice.

The default range for a standard scroll bar is 0 through 100. The default range for a scroll bar control is empty (both the nMinPos and nMaxPos parameter values are zero). The difference between the values specified by the nMinPos and nMaxPos parameters must not be greater than the value of MAXLONG.

Because the messages that indicate scroll bar position, WM_HSCROLL and WM_VSCROLL, are limited to 16 bits of position data, applications that rely solely on those messages for position data have a practical maximum value of 65,535 for the SetScrollRange function's nMaxPos parameter.

However, because the SetScrollInfo, SetScrollPos, SetScrollRange, GetScrollInfo, GetScrollPos, and GetScrollRange functions support 32-bit scroll bar position data, there is a way to circumvent the 16-bit barrier of the WM_HSCROLL and WM_VSCROLL messages. See GetScrollInfo for a description of the technique.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.391 SetSysColors

The SetSysColors function sets the colors for one or more display elements. Display elements are the various parts of a window and the display that appear on the system display screen.

```
SetSysColors: procedure
(
        cElements       :dword;
    var lpaElements     :var;
    var lpaRgbValues    :var
);
    @stdcall;
```

```
    @returns( "eax" );
    @external( "__imp__SetSysColors@12" );
```

Parameters

cElements

[in] Specifies the number of display elements in the array pointed to by the lpaElements parameter.

lpaElements

[in] Pointer to an array of integers that specify the display elements to be changed. For a list of display elements, see GetSysColor.

lpaRgbValues

[in] Pointer to an array of COLORREF values that contain the new red, green, blue (RGB) color values for the display elements in the array pointed to by the lpaElements parameter. To generate a COLORREF, use the RGB macro.

Return Values

If the function succeeds, the return value is a nonzero value.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

The SetSysColors function sends a WM_SYSCOLORCHANGE message to all windows to inform them of the change in color. It also directs the system to repaint the affected portions of all currently visible windows.

The SetSysColors function changes the current session only. The new colors are not saved when the system terminates.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.392 SetSystemCursor

The SetSystemCursor function enables an application to customize the system cursors. It replaces the contents of the system cursor specified by the id parameter with the contents of the cursor specified by the hcur parameter and then destroys hcur.

```
SetSystemCursor: procedure
(
    hcur        :dword;
    id          :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__SetSystemCursor@8" );
```

Parameters

hcur

[in] Handle to a cursor. The function replaces the contents of the system cursor specified by id with the contents of the cursor handled by hcur.

The system destroys hcur by calling the DestroyCursor function. Therefore, hcur cannot be a cursor loaded using

the LoadCursor function. To specify a cursor loaded from a resource, copy the cursor using the CopyCursor function, then pass the copy to SetSystemCursor.

id

[in] Specifies the system cursor to replace with the contents of hcur. This parameter can be one of the following values.

Value	Meaning
OCR_APPSTARTING	Standard arrow and small hourglass
OCR_NORMAL	Standard arrow
OCR_CROSS	Crosshair
OCR_HAND	Windows 2000: Hand
OCR_HELP	Arrow and question mark
OCR_IBEAM	I-beam
OCR_NO	Slashed circle
OCR_SIZEALL	Four-pointed arrow pointing north, south, east, and west
OCR_SIZENESW	Double-pointed arrow pointing northeast and southwest
OCR_SIZENS	Double-pointed arrow pointing north and south
OCR_SIZENWSE	Double-pointed arrow pointing northwest and southeast
OCR_SIZEWE	Double-pointed arrow pointing west and east
OCR_UP	Vertical arrow
OCR_WAIT	Hourglass

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Requirements

Windows NT/2000: Requires Windows NT 3.5 or later.

Windows 95/98: Requires Windows 95 or later.

3.393 SetThreadDesktop

The SetThreadDesktop function assigns a desktop to the calling thread. All subsequent operations on the desktop use the access rights granted to hDesktop.

```
SetThreadDesktop: procedure
(
    hDesktop        :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__SetThreadDesktop@4" );
```

Parameters

hDesktop

[in] Handle to the desktop to be assigned to the calling thread. This handle is returned by the CreateDesktop, GetThreadDesktop, and OpenDesktop functions.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

The SetThreadDesktop function will fail if the calling thread has any windows or hooks on its current desktop (unless the hDesktop parameter is a handle to the current desktop).

Requirements

Windows NT/2000: Requires Windows NT 3.51 or later.

Windows 95/98: Requires Windows 95 or later.

3.394 SetTimer

The SetTimer function creates a timer with the specified time-out value.

```
SetTimer: procedure
(
    hWnd        :dword;
    uIDEvent    :dword;
    uElapse     :dword;
    lpTimerFunc :TIMERPROC
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__SetTimer@16" );
```

Parameters

hWnd

[in] Handle to the window to be associated with the timer. This window must be owned by the calling thread. If this parameter is NULL, no window is associated with the timer and the nIDEvent parameter is ignored.

nIDEvent

[in] Specifies a nonzero timer identifier. If the hWnd parameter is NULL, this parameter is ignored.

If the hWnd parameter is not NULL and the window specified by hWnd already has a timer with the value nIDEvent, then the existing timer is replaced by the new timer. When SetTimer replaces a timer, the timer is reset. Therefore, a message will be sent after the current time-out value elapses, but the previously set time-out value is ignored.

uElapse

[in] Specifies the time-out value, in milliseconds.

lpTimerFunc

[in] Pointer to the function to be notified when the time-out value elapses. For more information about the function, see TimerProc.

If lpTimerFunc is NULL, the system posts a WM_TIMER message to the application queue. The hwnd member of the message's MSG structure contains the value of the hWnd parameter.

Return Values

If the function succeeds and the hWnd parameter is NULL, the return value is an integer identifying the new timer. An application can pass this value to the KillTimer function to destroy the timer.

If the function succeeds and the hWnd parameter is not NULL, then the return value is a nonzero integer. An application can pass the value of the nIDTimer parameter to the KillTimer function to destroy the timer.

If the function fails to create a timer, the return value is zero. To get extended error information, call GetLastEr-

ror.

Remarks

An application can process WM_TIMER messages by including a WM_TIMER case statement in the window procedure or by specifying a TimerProc callback function when creating the timer. When you specify a TimerProc callback function, the default window procedure calls the callback function when it processes WM_TIMER. Therefore, you need to dispatch messages in the calling thread, even when you use TimerProc instead of processing WM_TIMER.

The wParam parameter of the WM_TIMER message contains the value of the nIDEvent parameter.

The timer identifier, nIDEvent, is specific to the associated window. Another window can have its own timer which has the same identifier as a timer owned by another window. The timers are distinct.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.395 SetUserObjectInformation

The SetUserObjectInformation function sets information about a window station or desktop object.

```
SetUserObjectInformation: procedure
(
        hObj        :dword;
        nIndex      :dword;
    var pvInfo      :var;
        nLength     :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__SetUserObjectInformationA@16" );
```

Parameters

hObj

[in] Handle to the window station or desktop object for which to set object information. This value can be an HDESK or HWINSTA handle (for example, a handle returned by CreateWindowStation, OpenWindowStation, CreateDesktop, or OpenDesktop).

nIndex

[in] Specifies the object information to be set. This parameter must be the following value.

Value	Description
UOI_FLAGS	Sets the object's handle flags. The *pvInfo parameter must point to a USEROBJECTFLAGS structure.*

pvInfo

[in] Pointer to a buffer containing the object information.

nLength

[in] Specifies the size, in bytes, of the information contained in the buffer pointed to by pvInfo.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails the return value is zero. To get extended error information, call GetLastError.

Requirements

Windows NT/2000: Requires Windows NT 3.51 or later.

Windows 95/98: Unsupported.

3.396 SetWindowContextHelpId

Associates a Help context identifier with the specified window.

```
SetWindowContextHelpId: procedure
(
    hwnd            :dword;
    dwContextHelpID :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__SetWindowContextHelpId@8" );
```

Parameters

hwnd

Handle to the window with which to associate the Help context identifier.

dwContextHelpId

Help context identifier.

Return Values

Returns nonzero if successful, or zero otherwise.

To get extended error information, call GetLastError.

Remarks

If a child window does not have a Help context identifier, it inherits the identifier of its parent window. Likewise, if an owned window does not have a Help context identifier, it inherits the identifier of its owner window. This inheritance of Help context identifiers allows an application to set just one identifier for a dialog box and all of its controls.

See Also

GetWindowContextHelpId

Requirements

Windows NT/2000: Requires Windows NT 3.51 or later.

Windows 95/98: Requires Windows 95 or later.

3.397 SetWindowLong

The SetWindowLong function changes an attribute of the specified window. The function also sets the 32-bit (long) value at the specified offset into the extra window memory.

Note This function has been superseded by the SetWindowLongPtr function. To write code that is compatible with both 32-bit and 64-bit versions of Windows, use SetWindowLongPtr.

```
SetWindowLong: procedure
(
    hWnd            :dword;
    nIndex          :dword;
    dwNewLong       :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__SetWindowLongA@12" );
```

Parameters

hWnd

[in] Handle to the window and, indirectly, the class to which the window belongs.

Windows 95/98: The SetWindowLong function may fail if the window specified by the hWnd parameter does not belong to the same process as the calling thread.

nIndex

[in] Specifies the zero-based offset to the value to be set. Valid values are in the range zero through the number of bytes of extra window memory, minus 4; for example, if you specified 12 or more bytes of extra memory, a value of 8 would be an index to the third 32-bit integer. To set any other value, specify one of the following values.

Value	Action
GWL_EXSTYLE	Sets a new extended window style. For more information, see CreateWindowEx.
GWL_STYLE	Sets a new window style.
GWL_WNDPROC	Sets a new address for the window procedure. Windows NT/2000: You cannot change this attribute if the window does not belong to the same process as the calling thread.
GWL_HINSTANCE	Sets a new application instance handle.
GWL_ID	Sets a new identifier of the window.
GWL_USERDATA	Sets the 32-bit value associated with the window. Each window has a corresponding 32-bit value intended for use by the application that created the window. This value is initially zero.

The following values are also available when the hWnd parameter identifies a dialog box.

Value	Action
DWL_DLGPROC	Sets the new address of the dialog box procedure.
DWL_MSGRESULT	Sets the return value of a message processed in the dialog box procedure.
DWL_USER	Sets new extra information that is private to the application, such as handles or pointers.

dwNewLong

[in] Specifies the replacement value.

Return Values

If the function succeeds, the return value is the previous value of the specified 32-bit integer.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

If the previous value of the specified 32-bit integer is zero, and the function succeeds, the return value is zero, but the function does not clear the last error information. This makes it difficult to determine success or failure. To deal with this, you should clear the last error information by calling SetLastError(0) before calling SetWindowLong. Then, function failure will be indicated by a return value of zero and a GetLastError result that is nonzero.

Remarks

Certain window data is cached, so changes you make using SetWindowLong will not take effect until you call the SetWindowPos function. Specifically, if you change any of the frame styles, you must call SetWindowPos with the SWP_FRAMECHANGED flag for the cache to be updated properly.

If you use SetWindowLong with the GWL_WNDPROC index to replace the window procedure, the window procedure must conform to the guidelines specified in the description of the WindowProc callback function.

If you use SetWindowLong with the DWL_MSGRESULT index to set the return value for a message processed by a dialog procedure, you should return TRUE directly afterwards. Otherwise, if you call any function that results in your dialog procedure receiving a window message, the nested window message could overwrite the return value you set using DWL_MSGRESULT.

Calling SetWindowLong with the GWL_WNDPROC index creates a subclass of the window class used to create the window. An application can subclass a system class, but should not subclass a window class created by another process. The SetWindowLong function creates the window subclass by changing the window procedure associated with a particular window class, causing the system to call the new window procedure instead of the previous one. An application must pass any messages not processed by the new window procedure to the previous window procedure by calling CallWindowProc. This allows the application to create a chain of window procedures.

Reserve extra window memory by specifying a nonzero value in the cbWndExtra member of the WNDCLASSEX structure used with the RegisterClassEx function.

You must not call SetWindowLong with the GWL_HWNDPARENT index to change the parent of a child window. Instead, use the SetParent function.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.398 SetWindowPlacement

The SetWindowPlacement function sets the show state and the restored, minimized, and maximized positions of the specified window.

```
SetWindowPlacement: procedure
(
        hWnd        :dword;
    var lpwndpl     :WINDOWPLACEMENT
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__SetWindowPlacement@8" );
```

Parameters

hWnd

[in] Handle to the window.

lpwndpl

[in] Pointer to a WINDOWPLACEMENT structure that specifies the new show state and window positions.

Before calling SetWindowPlacement, set the length member of the WINDOWPLACEMENT structure to sizeof(WINDOWPLACEMENT). SetWindowPlacement fails if lpwndpl->length is not set correctly.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

If the information specified in WINDOWPLACEMENT would result in a window that is completely off the screen, the system will automatically adjust the coordinates so that the window is visible, taking into account changes in screen resolution and multiple monitor configuration.

The length member of WINDOWPLACEMENT must be set to sizeof(WINDOWPLACEMENT). If this member is not set correctly, the function returns FALSE. For additional remarks on the proper use of window placement coordinates, see WINDOWPLACEMENT.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.399 SetWindowPos

The SetWindowPos function changes the size, position, and Z order of a child, pop-up, or top-level window. Child, pop-up, and top-level windows are ordered according to their appearance on the screen. The topmost window receives the highest rank and is the first window in the Z order.

```
SetWindowPos: procedure
(
    hWnd            :dword;
    hWndInsertAfter :dword;
    X               :dword;
    Y               :dword;
    cx              :dword;
    cy              :dword;
    uFlags          :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__SetWindowPos@28" );
```

Parameters

hWnd

[in] Handle to the window.

hWndInsertAfter

[in] Handle to the window to precede the positioned window in the Z order. This parameter must be a window handle or one of the following values.

Value	Meaning
HWND_BOTTOM	Places the window at the bottom of the Z order. If the hWnd parameter identifies a topmost window, the window loses its topmost status and is placed at the bottom of all other windows.
HWND_NOTOPMOST	Places the window above all non-topmost windows (that is, behind all topmost windows). This flag has no effect if the window is already a non-topmost window.
HWND_TOP	Places the window at the top of the Z order.

HWND_TOPMOST	Places the window above all non-topmost windows. The window maintains its topmost position even when it is deactivated.

For more information about how this parameter is used, see the following Remarks section.

X

[in] Specifies the new position of the left side of the window, in client coordinates.

Y

[in] Specifies the new position of the top of the window, in client coordinates.

cx

[in] Specifies the new width of the window, in pixels.

cy

[in] Specifies the new height of the window, in pixels.

uFlags

[in] Specifies the window sizing and positioning flags. This parameter can be a combination of the following values.

Value	Meaning
SWP_ASYNCWINDOWPOS	If the calling thread and the thread that owns the window are attached to different input queues, the system posts the request to the thread that owns the window. This prevents the calling thread from blocking its execution while other threads process the request.
SWP_DEFERERASE	Prevents generation of the WM_SYNCPAINT message.
SWP_DRAWFRAME	Draws a frame (defined in the window's class description) around the window.
SWP_FRAMECHANGED	Applies new frame styles set using the SetWindowLong function. Sends a WM_NCCALCSIZE message to the window, even if the window's size is not being changed. If this flag is not specified, WM_NCCALCSIZE is sent only when the window's size is being changed.
SWP_HIDEWINDOW	Hides the window.
SWP_NOACTIVATE	Does not activate the window. If this flag is not set, the window is activated and moved to the top of either the top-most or non-topmost group (depending on the setting of the hWndInsertAfter parameter).
SWP_NOCOPYBITS	Discards the entire contents of the client area. If this flag is not specified, the valid contents of the client area are saved and copied back into the client area after the window is sized or repositioned.
SWP_NOMOVE	Retains the current position (ignores the X and Y parameters).
SWP_NOOWNERZORDER	Does not change the owner window's position in the Z order.
SWP_NOREDRAW	Does not redraw changes. If this flag is set, no repainting of any kind occurs. This applies to the client area, the non-client area (including the title bar and scroll bars), and any part of the parent window uncovered as a result of the window being moved. When this flag is set, the application must explicitly invalidate or redraw any parts of the window and parent window that need redrawing.

SWP_NOREPOSITION	Same as the SWP_NOOWNERZORDER flag.
SWP_NOSENDCHANGING	Prevents the window from receiving the WM_WINDOWPOSCHANGING message.
SWP_NOSIZE	Retains the current size (ignores the cx and cy parameters).
SWP_NOZORDER	Retains the current Z order (ignores the hWndInsertAfter parameter).
SWP_SHOWWINDOW	Displays the window.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

If the SWP_SHOWWINDOW or SWP_HIDEWINDOW flag is set, the window cannot be moved or sized.

If you have changed certain window data using SetWindowLong, you must call SetWindowPos to have the changes take effect. Use the following combination for uFlags: SWP_NOMOVE | SWP_NOSIZE | SWP_NOZORDER | SWP_FRAMECHANGED.

A window can be made a topmost window either by setting the hWndInsertAfter parameter to HWND_TOPMOST and ensuring that the SWP_NOZORDER flag is not set, or by setting a window's position in the Z order so that it is above any existing topmost windows. When a non-topmost window is made topmost, its owned windows are also made topmost. Its owners, however, are not changed.

If neither the SWP_NOACTIVATE nor SWP_NOZORDER flag is specified (that is, when the application requests that a window be simultaneously activated and its position in the Z order changed), the value specified in hWndInsertAfter is used only in the following circumstances:

- Neither the HWND_TOPMOST nor HWND_NOTOPMOST flag is specified in hWndInsertAfter.
- The window identified by hWnd is not the active window.

An application cannot activate an inactive window without also bringing it to the top of the Z order. Applications can change an activated window's position in the Z order without restrictions, or it can activate a window and then move it to the top of the topmost or non-topmost windows.

If a topmost window is repositioned to the bottom (HWND_BOTTOM) of the Z order or after any non-topmost window, it is no longer topmost. When a topmost window is made non-topmost, its owners and its owned windows are also made non-topmost windows.

A non-topmost window can own a topmost window, but the reverse cannot occur. Any window (for example, a dialog box) owned by a topmost window is itself made a topmost window, to ensure that all owned windows stay above their owner.

If an application is not in the foreground, and should be in the foreground, it must call the SetForegroundWindow function.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.400 SetWindowRgn

The SetWindowRgn function sets the window region of a window. The window region determines the area within the window where the system permits drawing. The system does not display any portion of a window that lies outside of the window region

```
SetWindowRgn: procedure
(
    hWnd        :dword;
    hRgn        :dword;
    bRedraw     :boolean
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__SetWindowRgn@12" );
```

Parameters

hWnd

[in] Handle to the window whose window region is to be set.

hRgn

[in] Handle to a region. The function sets the window region of the window to this region.

If hRgn is NULL, the function sets the window region to NULL.

bRedraw

[in] Specifies whether the system redraws the window after setting the window region. If bRedraw is TRUE, the system does so; otherwise, it does not.

Typically, you set bRedraw to TRUE if the window is visible.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Windows NT/ 2000: To get extended error information, call GetLastError.

Remarks

When this function is called, the system sends the WM_WINDOWPOSCHANGING and WM_WINDOWPOSCHANGED messages to the window.

The coordinates of a window's window region are relative to the upper-left corner of the window, not the client area of the window.

After a successful call to SetWindowRgn, the system owns the region specified by the region handle hRgn. The system does not make a copy of the region. Thus, you should not make any further function calls with this region handle. In particular, do not delete this region handle. The system deletes the region handle when it no longer needed.

To obtain the window region of a window, call the GetWindowRgn function.

Requirements

Windows NT/2000: Requires Windows NT 3.51 or later.

Windows 95/98: Requires Windows 95 or later.

3.401 SetWindowText

The SetWindowText function changes the text of the specified window's title bar (if it has one). If the specified window is a control, the text of the control is changed. However, SetWindowText cannot change the text of a control in another application.

```
SetWindowText: procedure
(
    hWnd        :dword;
    lpString    :string
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__SetWindowTextA@8" );
```

Parameters

hWnd

[in] Handle to the window or control whose text is to be changed.

lpString

[in] Pointer to a null-terminated string to be used as the new title or control text.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

If the target window is owned by the current process, SetWindowText causes a WM_SETTEXT message to be sent to the specified window or control. If the control is a list box control created with the WS_CAPTION style, however, SetWindowText sets the text for the control, not for the list box entries.

To set the text of a control in another process, send the WM_SETTEXT message directly instead of calling SetWindowText.

The SetWindowText function does not expand tab characters (ASCII code 0x09). Tab characters are displayed as vertical bar (|) characters.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.402 SetWindowWord

The SetWindowWord function is obsolete. It is provided only for compatibility with 16-bit versions of Windows. Win32-based applications should use the SetWindowLong function.

See Also

Window Classes Overview, Window Class Functions

3.403 SetWindowsHook

The SetWindowsHook function is obsolete. It is provided only for compatibility with 16-bit versions of Windows. Win32-based applications should use the SetWindowsHookEx function.

See Also

Hooks Overview, Hook Functions

3.404 SetWindowsHookEx

The SetWindowsHookEx function installs an application-defined hook procedure into a hook chain. You would install a hook procedure to monitor the system for certain types of events. These events are associated either with a specific thread or with all threads in the same desktop as the calling thread.

```
SetWindowsHookEx: procedure
(
    idHook      :dword;
    lpfn        :HOOKPROC;
    hMod        :dword;
    dwThreadID  :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__SetWindowsHookExA@16" );
```

Parameters

idHook

[in] Specifies the type of hook procedure to be installed. This parameter can be one of the following values.

Value	Description
WH_CALLWNDPROC	Installs a hook procedure that monitors messages before the system sends them to the destination window procedure. For more information, see the CallWndProc hook procedure.
WH_CALLWNDPROCRET	Installs a hook procedure that monitors messages after they have been processed by the destination window procedure. For more information, see the CallWndRetProc hook procedure.
WH_CBT	Installs a hook procedure that receives notifications useful to a computer-based training (CBT) application. For more information, see the CBTProc hook procedure.
WH_DEBUG	Installs a hook procedure useful for debugging other hook procedures. For more information, see the DebugProc hook procedure.
WH_FOREGROUNDIDLE	Installs a hook procedure that will be called when the application's foreground thread is about to become idle. This hook is useful for performing low priority tasks during idle time. For more information, see the ForegroundIdleProc hook procedure.
WH_GETMESSAGE	Installs a hook procedure that monitors messages posted to a message queue. For more information, see the GetMsgProc hook procedure.
WH_JOURNALPLAYBACK	Installs a hook procedure that posts messages previously recorded by a WH_JOURNALRECORD hook procedure. For more information, see the JournalPlaybackProc hook procedure.
WH_JOURNALRECORD	Installs a hook procedure that records input messages posted to the system message queue. This hook is useful for recording macros. For more information, see the JournalRecordProc hook procedure.

WH_KEYBOARD	Installs a hook procedure that monitors keystroke messages. For more information, see the KeyboardProc hook procedure.
WH_KEYBOARD_LL	Windows NT/2000: Installs a hook procedure that monitors low-level keyboard input events. For more information, see the LowLevelKeyboardProc hook procedure.
WH_MOUSE	Installs a hook procedure that monitors mouse messages. For more information, see the MouseProc hook procedure.
WH_MOUSE_LL	Windows NT/2000: Installs a hook procedure that monitors low-level mouse input events. For more information, see the LowLevelMouseProc hook procedure.
WH_MSGFILTER	Installs a hook procedure that monitors messages generated as a result of an input event in a dialog box, message box, menu, or scroll bar. For more information, see the MessageProc hook procedure.
WH_SHELL	Installs a hook procedure that receives notifications useful to shell applications. For more information, see the ShellProc hook procedure.
WH_SYSMSGFILTER	Installs a hook procedure that monitors messages generated as a result of an input event in a dialog box, message box, menu, or scroll bar. The hook procedure monitors these messages for all applications in the same desktop as the calling thread. For more information, see the SysMsgProc hook procedure.

lpfn

[in] Pointer to the hook procedure. If the dwThreadId parameter is zero or specifies the identifier of a thread created by a different process, the lpfn parameter must point to a hook procedure in a dynamic-link library (DLL). Otherwise, lpfn can point to a hook procedure in the code associated with the current process.

hMod

[in] Handle to the DLL containing the hook procedure pointed to by the lpfn parameter. The hMod parameter must be set to NULL if the dwThreadId parameter specifies a thread created by the current process and if the hook procedure is within the code associated with the current process.

dwThreadId

[in] Specifies the identifier of the thread with which the hook procedure is to be associated. If this parameter is zero, the hook procedure is associated with all existing threads running in the same desktop as the calling thread.

Return Values

If the function succeeds, the return value is the handle to the hook procedure.

If the function fails, the return value is NULL. To get extended error information, call GetLastError.

Remarks

An error may occur if the hMod parameter is NULL and the dwThreadId parameter is zero or specifies the identifier of a thread created by another process.

Calling the CallNextHookEx function to chain to the next hook procedure is optional, but it is highly recommended; otherwise, other applications that have installed hooks will not receive hook notifications and may behave incorrectly as a result. You should call CallNextHookEx unless you absolutely need to prevent the notification from being seen by other applications.

Before terminating, an application must call the UnhookWindowsHookEx function to free system resources associated with the hook.

The scope of a hook depends on the hook type. Some hooks can be set only with global scope; others can also be set for only a specific thread, as shown in the following table.

Hook	Scope
WH_CALLWNDPROC	Thread or global
WH_CALLWNDPROCRET	Thread or global
WH_CBT	Thread or global
WH_DEBUG	Thread or global
WH_FOREGROUNDIDLE	Thread or global
WH_GETMESSAGE	Thread or global
WH_JOURNALPLAYBACK	Global only
WH_JOURNALRECORD	Global only
WH_KEYBOARD	Thread or global
WH_KEYBOARD_LL	Global only
WH_MOUSE	Thread or global
WH_MOUSE_LL	Global only
WH_MSGFILTER	Thread or global
WH_SHELL	Thread or global
WH_SYSMSGFILTER	Global only

For a specified hook type, thread hooks are called first, then global hooks.

The global hooks are a shared resource, and installing one affects all applications in the same desktop as the calling thread. All global hook functions must be in libraries. Global hooks should be restricted to special-purpose applications or to use as a development aid during application debugging. Libraries that no longer need a hook should remove its hook procedure.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.405 ShowCaret

The ShowCaret function makes the caret visible on the screen at the caret's current position. When the caret becomes visible, it begins flashing automatically.

```
ShowCaret: procedure
(
    hWnd        :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__ShowCaret@4" );
```

Parameters

hWnd

[in] Handle to the window that owns the caret. If this parameter is NULL, ShowCaret searches the current task for the window that owns the caret.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

ShowCaret shows the caret only if the specified window owns the caret, the caret has a shape, and the caret has not been hidden two or more times in a row. If one or more of these conditions is not met, ShowCaret does nothing and returns FALSE.

Hiding is cumulative. If your application calls HideCaret five times in a row, it must also call ShowCaret five times before the caret reappears.

The system provides one caret per queue. A window should create a caret only when it has the keyboard focus or is active. The window should destroy the caret before losing the keyboard focus or becoming inactive.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.406 ShowCursor

The ShowCursor function displays or hides the cursor.

```
ShowCursor: procedure
(
    bShow       :boolean
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__ShowCursor@4" );
```

Parameters

bShow

[in] Specifies whether the internal display counter is to be incremented or decremented. If bShow is TRUE, the display count is incremented by one. If bShow is FALSE, the display count is decremented by one.

Return Values

The return value specifies the new display counter.

Remarks

This function sets an internal display counter that determines whether the cursor should be displayed. The cursor is displayed only if the display count is greater than or equal to 0. If a mouse is installed, the initial display count is 0. If no mouse is installed, the display count is –1.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.407 ShowOwnedPopups

The ShowOwnedPopups function shows or hides all pop-up windows owned by the specified window.

```
ShowOwnedPopups: procedure
(
    hWnd        :dword;
    fShow       :dword
```

```
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__ShowOwnedPopups@8" );
```

Parameters

hWnd

[in] Handle to the window that owns the pop-up windows to be shown or hidden.

fShow

[in] Specifies whether pop-up windows are to be shown or hidden. If this parameter is TRUE, all hidden pop-up windows are shown. If this parameter is FALSE, all visible pop-up windows are hidden.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

ShowOwnedPopups shows only windows hidden by a previous call to ShowOwnedPopups. For example, if a pop-up window is hidden by using the ShowWindow function, subsequently calling ShowOwnedPopups with the fShow parameter set to TRUE does not cause the window to be shown.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.408 ShowScrollBar

The ShowScrollBar function shows or hides the specified scroll bar.

```
ShowScrollBar: procedure
(
    hWnd        :dword;
    wBar        :dword;
    bShow       :boolean
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__ShowScrollBar@12" );
```

Parameters

hWnd

[in] Handle to a scroll bar control or a window with a standard scroll bar, depending on the value of the wBar parameter.

wBar

[in] Specifies the scroll bar(s) to be shown or hidden. This parameter can be one of the following values.

Value	Meaning
SB_BOTH	Shows or hides a window's standard horizontal and vertical scroll bars.

SB_CTL	Shows or hides a scroll bar control. The hWnd parameter must be the handle to the scroll bar control.
SB_HORZ	Shows or hides a window's standard horizontal scroll bars.
SB_VERT	Shows or hides a window's standard vertical scroll bar.

bShow

[in] Specifies whether the scroll bar is shown or hidden. If this parameter is TRUE, the scroll bar is shown; otherwise, it is hidden.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

You should not call this function to hide a scroll bar while processing a scroll bar message.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.409 ShowWindow

The ShowWindow function sets the specified window's show state.

```
ShowWindow: procedure
(
    hWnd        :dword;
    nCmdShow    :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__ShowWindow@8" );
```

Parameters

hWnd

[in] Handle to the window.

nCmdShow

[in] Specifies how the window is to be shown. This parameter is ignored the first time an application calls ShowWindow, if the program that launched the application provides a STARTUPINFO structure. Otherwise, the first time ShowWindow is called, the value should be the value obtained by the WinMain function in its nCmdShow parameter. In subsequent calls, this parameter can be one of the following values.

Value	Meaning
SW_FORCEMINIMIZE	Windows 2000: Minimizes a window, even if the thread that owns the window is hung. This flag should only be used when minimizing windows from a different thread.
SW_HIDE	Hides the window and activates another window.
SW_MAXIMIZE	Maximizes the specified window.
SW_MINIMIZE	Minimizes the specified window and activates the next top-level window in the Z order.

SW_RESTORE	Activates and displays the window. If the window is minimized or maximized, the system restores it to its original size and position. An application should specify this flag when restoring a minimized window.
SW_SHOW	Activates the window and displays it in its current size and position.
SW_SHOWDEFAULT	Sets the show state based on the SW_ value specified in the STARTUPINFO structure passed to the CreateProcess function by the program that started the application.
SW_SHOWMAXIMIZED	Activates the window and displays it as a maximized window.
SW_SHOWMINIMIZED	Activates the window and displays it as a minimized window.
SW_SHOWMINNOACTIVE	Displays the window as a minimized window. This value is similar to SW_SHOWMINIMIZED, except the window is not activated.
SW_SHOWNA	Displays the window in its current size and position. This value is similar to SW_SHOW, except the window is not activated.
SW_SHOWNOACTIVATE	Displays a window in its most recent size and position. This value is similar to SW_SHOWNORMAL, except the window is not actived.
SW_SHOWNORMAL	Activates and displays a window. If the window is minimized or maximized, the system restores it to its original size and position. An application should specify this flag when displaying the window for the first time.

Return Values

If the window was previously visible, the return value is nonzero.

If the window was previously hidden, the return value is zero.

Remarks

The first time an application calls ShowWindow, it should use the WinMain function's nCmdShow parameter as its nCmdShow parameter. Subsequent calls to ShowWindow must use one of the values in the given list, instead of the one specified by the WinMain function's nCmdShow parameter.

As noted in the discussion of the nCmdShow parameter, the nCmdShow value is ignored in the first call to ShowWindow if the program that launched the application specifies startup information in the STARTUPINFO structure. In this case, ShowWindow uses the information specified in the STARTUPINFO structure to show the window. On subsequent calls, the application must call ShowWindow with nCmdShow set to SW_SHOWDEFAULT to use the startup information provided by the program that launched the application. This behavior is designed for the following situations:

- Applications create their main window by calling CreateWindow with the WS_VISIBLE flag set.
- Applications create their main window by calling CreateWindow with the WS_VISIBLE flag cleared, and later call ShowWindow with the SW_SHOW flag set to make it visible.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.410 ShowWindowAsync

The ShowWindowAsync function sets the show state of a window created by a different thread.

```
ShowWindowAsync: procedure
(
    hWnd        :dword;
    nCmdShow    :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__ShowWindowAsync@8" );
```

Parameters

hWnd

[in] Handle to the window.

nCmdShow

[in] Specifies how the window is to be shown. For a list of possible values, see the description of the ShowWindow function.

Return Values

If the window was previously visible, the return value is nonzero.

If the window was previously hidden, the return value is zero.

Remarks

This function posts a show-window event to the message queue of the given window. An application can use this function to avoid becoming hung while waiting for a hung application to finish processing a show-window event.

Requirements

Windows NT/2000: Requires Windows NT 4.0 or later.

Windows 95/98: Requires Windows 95 or later.

3.411 SubtractRect

The SubtractRect function determines the coordinates of a rectangle formed by subtracting one rectangle from another.

```
SubtractRect: procedure
(
    var lprcDst     :RECT;
    var lprcSrc1    :RECT;
    var lprcSrc2    :RECT
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__SubtractRect@12" );
```

Parameters

lprcDst

[out] Pointer to a RECT structure that receives the coordinates of the rectangle determined by subtracting the rectangle pointed to by lprcSrc2 from the rectangle pointed to by lprcSrc1.

lprcSrc1

[in] Pointer to a RECT structure from which the function subtracts the rectangle pointed to by lprcSrc2.

lprcSrc2

[in] Pointer to a RECT structure that the function subtracts from the rectangle pointed to by lprcSrc1.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Windows NT/ 2000: To get extended error information, call GetLastError.

Remarks

The function only subtracts the rectangle specified by lprcSrc2 from the rectangle specified by lprcSrc1 when the rectangles intersect completely in either the x- or y-direction. For example, if *lprcSrc1 has the coordinates (10,10,100,100) and *lprcSrc2 has the coordinates (50,50,150,150), the function sets the coordinates of the rectangle pointed to by lprcDst to (10,10,100,100). If *lprcSrc1 has the coordinates (10,10,100,100) and *lprcSrc2 has the coordinates (50,10,150,150), however, the function sets the coordinates of the rectangle pointed to by lprcDst to (10,10,50,100).

Requirements

Windows NT/2000: Requires Windows NT 3.5 or later.

Windows 95/98: Requires Windows 95 or later.

3.412 SwapMouseButton

The SwapMouseButton function reverses or restores the meaning of the left and right mouse buttons.

```
SwapMouseButton: procedure
(
    fSwap   :boolean
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__SwapMouseButton@4" );
```

Parameters

fSwap

[in] Specifies whether the mouse button meanings are reversed or restored. If this parameter is TRUE, the left button generates right-button messages and the right button generates left-button messages. If this parameter is FALSE, the buttons are restored to their original meanings.

Return Values

If the meaning of the mouse buttons was reversed previously, before the function was called, the return value is nonzero.

If the meaning of the mouse buttons was not reversed, the return value is zero.

Remarks

Button swapping is provided as a convenience to people who use the mouse with their left hands. The SwapMouseButton function is usually called by Control Panel only. Although an application is free to call the function, the mouse is a shared resource and reversing the meaning of its buttons affects all applications.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.413 SwitchDesktop

The SwitchDesktop function makes a desktop visible and activates it. This enables the desktop to receive input from the user. The calling process must have DESKTOP_SWITCHDESKTOP access to the desktop for the SwitchDesktop function to succeed.

```
SwitchDesktop: procedure
(
    hDesktop        :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__SwitchDesktop@4" );
```

Parameters

hDesktop

[in] Handle to the desktop to be made visible and active. This handle is returned by the CreateDesktop and OpenDesktop functions.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

The SwitchDesktop function fails if the desktop belongs to an invisible window station. SwitchDesktop also fails when called from a process associated with a secured desktop, such as the "WinLogon" and "Screen-Saver" desktops.

Requirements

Windows NT/2000: Requires Windows NT 3.51 or later.

Windows 95/98: Unsupported.

3.414 SystemParametersInfo

The SystemParametersInfo function retrieves or sets the value of one of the system-wide parameters. This function can also update the user profile while setting a parameter.

```
SystemParametersInfo: procedure
(
        uiAction    :dword;
        uiParam     :dword;
    var pvParam     :var;
        fWinIni     :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__SystemParametersInfoA@16" );
```

Parameters

uiAction

[in] Specifies the system-wide parameter to retrieve or set. This parameter can be one of the values from the following tables.

The following are the accessibility parameters.

Accessibility parameter	Meaning
SPI_GETACCESSTIMEOUT	Retrieves information about the time-out period associated with the accessibility features. The *pvParam parameter must point to an ACCESSTIMEOUT structure that receives the information. Set the cbSize member of this structure and the uiParam parameter to sizeof(ACCESSTIMEOUT).*
SPI_GETFILTERKEYS	Retrieves information about the FilterKeys accessibility feature. The pvParam parameter must point to a FILTERKEYS structure that receives the information. Set the cbSize member of this structure and the uiParam parameter to sizeof(FILTERKEYS).
SPI_GETFOCUSBORDERHEIGHT	Whistler: Gets the height, in pixels, of the top and bottom edges of the focus rectangle drawn with DrawFocusRect. The pvParam parameter must point to a UINT.
SPI_GETFOCUSBORDERWIDTH	Whistler: Gets the width, in pixels, of the left and right edges of the focus rectangle drawn with DrawFocusRect. The pvParam parameter must point to a UINT.
SPI_GETHIGHCONTRAST	Windows 95/98, Windows 2000: Retrieves information about the HighContrast accessibility feature. The pvParam parameter must point to a HIGHCONTRAST structure that receives the information. Set the cbSize member of this structure and the uiParam parameter to sizeof(HIGHCONTRAST).
SPI_GETMOUSECLICKLOCK	Windows Me and Whistler: Gets the state of the Mouse ClickLock feature. The pvParam parameter must point to a BOOL variable that receives TRUE if enabled, or FALSE otherwise.
SPI_GETMOUSECLICKLOCKTIME	Windows Me and Whistler: Gets the time delay before the primary mouse button is locked. The pvParam parameter must point to DWORD that receives the time delay. This is only enabled if SPI_SETMOUSECLICKLOCK is set to TRUE.
SPI_GETMOUSEKEYS	Retrieves information about the MouseKeys accessibility feature. The pvParam parameter must point to a MOUSEKEYS structure that receives the information. Set the cbSize member of this structure and the uiParam parameter to sizeof(MOUSEKEYS).

SPI_GETMOUSESONAR	Windows Me and Whistler: Gets the state of the Mouse Sonar feature. The pvParam parameter must point to a BOOL variable that receives TRUE if enabled or FALSE otherwise.
SPI_GETMOUSEVANISH	Windows Me and Whistler: Gets the state of the Mouse Vanish feature. The pvParam parameter must point to a BOOL variable that receives TRUE if enabled or FALSE otherwise.
SPI_GETSCREENREADER	Windows 95/98, Windows 2000: Determines whether a screen reviewer utility is running. A screen reviewer utility directs textual information to an output device, such as a speech synthesizer or Braille display. When this flag is set, an application should provide textual information in situations where it would otherwise present the information graphically. The pvParam parameter is a pointer to a BOOL variable that receives TRUE if a screen reviewer utility is running, or FALSE otherwise.
SPI_GETSERIALKEYS	Windows 95/98: Retrieves information about the SerialKeys accessibility feature. The pvParam parameter must point to a SERIALKEYS structure that receives the information. Set the cbSize member of this structure and the uiParam parameter to sizeof(SERIALKEYS). Windows NT/2000: Not supported.
SPI_GETSHOWSOUNDS	Determines whether the Show Sounds accessibility flag is on or off. If it is on, the user requires an application to present information visually in situations where it would otherwise present the information only in audible form. The pvParam parameter must point to a BOOL variable that receives TRUE if the feature is on, or FALSE if it is off. Using this value is equivalent to calling GetSystemMetrics (SM_SHOWSOUNDS). That is the recommended call.
SPI_GETSOUNDSENTRY	Retrieves information about the SoundSentry accessibility feature. The pvParam parameter must point to a SOUNDSENTRY structure that receives the information. Set the cbSize member of this structure and the uiParam parameter to sizeof(SOUNDSENTRY).
SPI_GETSTICKYKEYS	Retrieves information about the StickyKeys accessibility feature. The pvParam parameter must point to a STICKYKEYS structure that receives the information. Set the cbSize member of this structure and the uiParam parameter to sizeof(STICKYKEYS).

SPI_GETTOGGLEKEYS	Retrieves information about the ToggleKeys accessibility feature. The pvParam parameter must point to a TOGGLEKEYS structure that receives the information. Set the cbSize member of this structure and the uiParam parameter to sizeof(TOGGLEKEYS).
SPI_SETACCESSTIMEOUT	Sets the time-out period associated with the accessibility features. The pvParam parameter must point to an ACCESSTIMEOUT structure that contains the new parameters. Set the cbSize member of this structure and the uiParam parameter to sizeof(ACCESSTIMEOUT).
SPI_SETFILTERKEYS	Sets the parameters of the FilterKeys accessibility feature. The pvParam parameter must point to a FILTERKEYS structure that contains the new parameters. Set the cbSize member of this structure and the uiParam parameter to sizeof(FILTERKEYS).
SPI_SETFOCUSBORDERHEIGHT	Whistler: Sets the height of the top and bottom edges of the focus rectangle drawn with DrawFocusRect to the value of the pvParam parameter.
SPI_SETFOCUSBORDERWIDTH	Whistler: Sets the height of the left and right edges of the focus rectangle drawn with DrawFocusRect to the value of the pvParam parameter.
SPI_SETHIGHCONTRAST	Windows 95/98, Windows 2000: Sets the parameters of the HighContrast accessibility feature. The pvParam parameter must point to a HIGHCONTRAST structure that contains the new parameters. Set the cbSize member of this structure and the uiParam parameter to sizeof(HIGHCONTRAST).
SPI_SETMOUSECLICKLOCK	Windows Me and Whistler: Turns the Mouse ClickLock accessibility feature on or off. This feature temporarily locks down the primary mouse button when that button is clicked and held down for the time specified by SPI_SETMOUSECLICKLOCKTIME. The uiParam parameter specifies TRUE for on, or FALSE for off. The default is off. For more information, see Remarks.
SPI_SETMOUSECLICKLOCKTIME	Windows Me and Whistler: Adjusts the time delay before the primary mouse button is locked. The uiParam parameter specifies the time delay in microseconds. For example, specify 1000 for a 1 second delay. The default is 1200.

SPI_SETMOUSEKEYS	Sets the parameters of the MouseKeys accessibility feature. The pvParam parameter must point to a MOUSEKEYS structure that contains the new parameters. Set the cbSize member of this structure and the uiParam parameter to sizeof(MOUSEKEYS).
SPI_SETMOUSESONAR	Windows Me and Whistler: Turns the Sonar accessibility feature on or off. This feature briefly shows several concentric circles around the mouse pointer when the user presses and releases the CTRL key. The uiParam parameter specifies TRUE for on and FALSE for off. The default is off.
SPI_SETMOUSEVANISH	Windows Me and Whistler: Turns the Vanish feature on or off. This feature hides the mouse pointer when the user types; the pointer reappears when the user moves the mouse. The uiParam parameter specifies TRUE for on and FALSE for off. The default is off.
SPI_SETSCREENREADER	Windows 95/98, Windows 2000: Indicates whether a screen review utility is running. The uiParam parameter specifies TRUE for on, or FALSE for off.
SPI_SETSERIALKEYS	Windows 95/98: Sets the parameters of the SerialKeys accessibility feature. The pvParam parameter must point to a SERIALKEYS structure that contains the new parameters. Set the cbSize member of this structure and the uiParam parameter to sizeof(SERIALKEYS). Windows NT/2000: Not supported.
SPI_SETSHOWSOUNDS	Sets the ShowSounds accessibility feature as on or off. The uiParam parameter specifies TRUE for on, or FALSE for off.
SPI_SETSOUNDSENTRY	Sets the parameters of the SoundSentry accessibility feature. The pvParam parameter must point to a SOUNDSENTRY structure that contains the new parameters. Set the cbSize member of this structure and the uiParam parameter to sizeof(SOUNDSENTRY).
SPI_SETSTICKYKEYS	Sets the parameters of the StickyKeys accessibility feature. The pvParam parameter must point to a STICKYKEYS structure that contains the new parameters. Set the cbSize member of this structure and the uiParam parameter to sizeof(STICKYKEYS).
SPI_SETTOGGLEKEYS	Sets the parameters of the ToggleKeys accessibility feature. The pvParam parameter must point to a TOGGLEKEYS structure that contains the new parameters. Set the cbSize member of this structure and the uiParam parameter to sizeof(TOGGLEKEYS).

The following are the desktop parameters.

Desktop parameter	Meaning
SPI_GETDESKWALLPAPER	Windows 2000: Retrieves the full path of the bitmap file for the desktop wallpaper. The *pvParam parameter must point to a buffer that receives a null-terminated path string. Set the uiParam parameter to the size, in characters, of the pvParam buffer. The returned string will not exceed MAX_PATH characters. If there is no desktop wallpaper, the returned string is empty.*
SPI_GETDROPSHADOW	Whistler: Indicates whether the drop shadow effect is enabled. The pvParam parameter must point to a BOOL variable that returns TRUE if enabled or FALSE if disabled.
SPI_GETFLATMENU	Whistler: Indicates whether native User menus have flat menu appearance. The pvParam parameter must point to a BOOL variable that returns TRUE if the flat menu appearance is set, or FALSE otherwise.
SPI_GETFONTSMOOTHING	Indicates whether the font smoothing feature is enabled. This feature uses font antialiasing to make font curves appear smoother by painting pixels at different gray levels. The pvParam parameter must point to a BOOL variable that receives TRUE if the feature is enabled, or FALSE if it is not. Windows 95: This flag is supported only if Windows Plus! is installed. See SPI_GETWINDOWSEXTENSION.
SPI_GETWORKAREA	Retrieves the size of the work area on the primary display monitor. The work area is the portion of the screen not obscured by the system taskbar or by application desktop toolbars. The pvParam parameter must point to a RECT structure that receives the coordinates of the work area, expressed in virtual screen coordinates. To get the work area of a monitor other than the primary display monitor, call the GetMonitorInfo function.
SPI_SETCURSORS	Reloads the system cursors. Set the uiParam parameter to zero and the pvParam parameter to NULL
SPI_SETDESKPATTERN	Sets the current desktop pattern by causing Windows to read the Pattern= setting from the WIN.INI file.

SPI_SETDESKWALLPAPER	Sets the desktop wallpaper. The value of the pvParam parameter determines the new wallpaper. To specify a wallpaper bitmap, set pvParam to point to a null-terminated string containing the name of a bitmap file. Setting pvParam to "" removes the wallpaper. Setting pvParam to SETWALLPAPER_DEFAULT or NULL reverts to the default wallpaper.
SPI_SETDROPSHADOW	Whistler: Enables or disables the drop shadow effect. Set pvParam to TRUE to enable the drop shadow effect or FALSE to disable it. You must also have CS_DROPSHADOW in the window class style.
SPI_SETFLATMENU	Whistler: Enables or disables flat menu appearance for native User menus. Set pvParam to TRUE to enable flat menu appearance or FALSE to disable it. When enabled, the menu bar uses COLOR_MENUBAR for the menubar background, COLOR_MENU for the menu-popup background, COLOR_MENUHILIGHT for the fill of the current menu selection, and COLOR_HILIGHT for the outline of the current menu selection. If disabled, menus are drawn using the same metrics and colors as in Windows 2000 and earlier.
SPI_SETFONTSMOOTHING	Enables or disables the font smoothing feature, which uses font antialiasing to make font curves appear smoother by painting pixels at different gray levels. To enable the feature, set the uiParam parameter to TRUE. To disable the feature, set uiParam to FALSE. Windows 95: This flag is supported only if Windows Plus! is installed. See SPI_GETWINDOWSEXTENSION.
SPI_SETWORKAREA	Sets the size of the work area. The work area is the portion of the screen not obscured by the system taskbar or by application desktop toolbars. The pvParam parameter is a pointer to a RECT structure that specifies the new work area rectangle, expressed in virtual screen coordinates. In a system with multiple display monitors, the function sets the work area of the monitor that contains the specified rectangle. If pvParam is NULL, the function sets the work area of the primary display monitor to the full screen.

The following are the icon parameters.

Icon parameter	Meaning

SPI_GETICONMETRICS	Retrieves the metrics associated with icons. The *pvParam parameter must point to an ICONMETRICS structure that receives the information. Set the cbSize member of this structure and the uiParam parameter to sizeof(ICONMETRICS).*
SPI_GETICONTITLELOGFONT	Retrieves the logical font information for the current icon-title font. The uiParam parameter specifies the size of a LOGFONT structure, and the pvParam parameter must point to the LOGFONT structure to fill in.
SPI_GETICONTITLEWRAP	Determines whether icon-title wrapping is enabled. The pvParam parameter must point to a BOOL variable that receives TRUE if enabled, or FALSE otherwise.
SPI_ICONHORIZONTALSPACING	Sets or retrieves the width, in pixels, of an icon cell. The system uses this rectangle to arrange icons in large icon view. To set this value, set uiParam to the new value and set pvParam to NULL. You cannot set this value to less than SM_CXICON. To retrieve this value, pvParam must point to an integer that receives the current value.
SPI_ICONVERTICALSPACING	Sets or retrieves the height, in pixels, of an icon cell. To set this value, set uiParam to the new value and set pvParam to NULL. You cannot set this value to less than SM_CYICON. To retrieve this value, pvParam must point to an integer that receives the current value.
SPI_SETICONMETRICS	Sets the metrics associated with icons. The pvParam parameter must point to an ICONMETRICS structure that contains the new parameters. Set the cbSize member of this structure and the uiParam parameter to sizeof(ICONMETRICS).
SPI_SETICONS	Reloads the system icons. Set the uiParam parameter to zero and the pvParam parameter to NULL
SPI_SETICONTITLELOGFONT	Sets the font that is used for icon titles. The uiParam parameter specifies the size of a LOGFONT structure, and the pvParam parameter must point to a LOGFONT structure.
SPI_SETICONTITLEWRAP	Turns icon-title wrapping on or off. The uiParam parameter specifies TRUE for on, or FALSE for off.

The following are the input parameters. They include parameters related to the keyboard, mouse, input language, and the warning beeper.

Input parameter	Meaning

SPI_GETBEEP	Indicates whether the warning beeper is on. The *pvParam parameter must point to a BOOL variable that receives TRUE if the beeper is on, or FALSE if it is off.*
SPI_GETDEFAULTINPUTLANG	Returns the input locale identifier for the system default input language. The pvParam parameter must point to an HKL variable that receives this value. For more information, see Languages, Locales, and Keyboard Layouts.
SPI_GETKEYBOARDCUES	Windows 98, Windows 2000: Indicates whether menu access keys are always underlined. The pvParam parameter must point to a BOOL variable that receives TRUE if menu access keys are always underlined, and FALSE if they are underlined only when the menu is activated by the keyboard.
SPI_GETKEYBOARDDELAY	Retrieves the keyboard repeat-delay setting, which is a value in the range from 0 (approximately 250 ms delay) through 3 (approximately 1 second delay). The actual delay associated with each value may vary depending on the hardware. The pvParam parameter must point to an integer variable that receives the setting.
SPI_GETKEYBOARDPREF	Windows 95/98, Windows 2000: Determines whether the user relies on the keyboard instead of the mouse, and wants applications to display keyboard interfaces that would otherwise be hidden. The pvParam parameter must point to a BOOL variable that receives TRUE if the user relies on the keyboard; or FALSE otherwise.
SPI_GETKEYBOARDSPEED	Retrieves the keyboard repeat-speed setting, which is a value in the range from 0 (approximately 2.5 repetitions per second) through 31 (approximately 30 repetitions per second). The actual repeat rates are hardware-dependent and may vary from a linear scale by as much as 20%. The pvParam parameter must point to a DWORD variable that receives the setting.
SPI_GETMOUSE	Retrieves the two mouse threshold values and the mouse acceleration. The pvParam parameter must point to an array of three integers that receives these values. See mouse_event for further information.
SPI_GETMOUSEHOVERHEIGHT	Windows NT 4.0, Windows 98, Windows 2000: Gets the height, in pixels, of the rectangle within which the mouse pointer has to stay for TrackMouseEvent to generate a WM_MOUSEHOVER message. The pvParam parameter must point to a UINT variable that receives the height.

SPI_GETMOUSEHOVERTIME	Windows NT 4.0, Windows 98, Windows 2000: Gets the time, in milliseconds, that the mouse pointer has to stay in the hover rectangle for TrackMouseEvent to generate a WM_MOUSEHOVER message. The pvParam parameter must point to a UINT variable that receives the time.
SPI_GETMOUSEHOVERWIDTH	Windows NT 4.0, Windows 98, Windows 2000: Gets the width, in pixels, of the rectangle within which the mouse pointer has to stay for TrackMouseEvent to generate a WM_MOUSEHOVER message. The pvParam parameter must point to a UINT variable that receives the width.
SPI_GETMOUSESPEED	Windows 98, Windows 2000: Retrieves the current mouse speed. The mouse speed determines how far the pointer will move based on the distance the mouse moves. The pvParam parameter must point to an integer that receives a value which ranges between 1 (slowest) and 20 (fastest). A value of 10 is the default. The value can be set by an end user using the mouse control panel application or by an application using SPI_SETMOUSESPEED.
SPI_GETMOUSETRAILS	Windows 95/98 and Whistler: Indicates whether the Mouse Trails feature is enabled. This feature improves the visibility of mouse cursor movements by briefly showing a trail of cursors and quickly erasing them. The pvParam parameter must point to an integer variable that receives a value. If the value is zero or 1, the feature is disabled. If the value is greater than 1, the feature is enabled and the value indicates the number of cursors drawn in the trail. The uiParam parameter is not used. Windows NT/2000: Not supported.
SPI_GETSNAPTODEFBUTTON	Windows NT 4.0, Windows 98, Windows 2000: Determines whether the snap-to-default-button feature is enabled. If enabled, the mouse cursor automatically moves to the default button, such as OK or Apply, of a dialog box. The pvParam parameter must point to a BOOL variable that receives TRUE if the feature is on, or FALSE if it is off.
SPI_GETWHEELSCROLLLINES	Windows NT 4.0, Windows 98, Windows 2000: Gets the number of lines to scroll when the mouse wheel is rotated. The pvParam parameter must point to a UINT variable that receives the number of lines. The default value is 3.

SPI_SETBEEP	Turns the warning beeper on or off. The uiParam parameter specifies TRUE for on, or FALSE for off.
SPI_SETDEFAULTINPUTLANG	Sets the default input language for the system shell and applications. The specified language must be displayable using the current system character set. The pvParam parameter must point to an HKL variable that contains the input locale identifier for the default language. For more information, see Languages, Locales, and Keyboard Layouts.
SPI_SETDOUBLECLICKTIME	Sets the double-click time for the mouse to the value of the uiParam parameter. The double-click time is the maximum number of milliseconds that can occur between the first and second clicks of a double-click. You can also call the SetDoubleClickTime function to set the double-click time. To get the current double-click time, call the GetDoubleClickTime function.
SPI_SETDOUBLECLKHEIGHT	Sets the height of the double-click rectangle to the value of the uiParam parameter. The double-click rectangle is the rectangle within which the second click of a double-click must fall for it to be registered as a double-click. To retrieve the height of the double-click rectangle, call GetSystemMetrics with the SM_CYDOUBLECLK flag.
SPI_SETDOUBLECLKWIDTH	Sets the width of the double-click rectangle to the value of the uiParam parameter. The double-click rectangle is the rectangle within which the second click of a double-click must fall for it to be registered as a double-click. To retrieve the width of the double-click rectangle, call GetSystemMetrics with the SM_CXDOUBLECLK flag.
SPI_SETKEYBOARDCUES	Windows 98, Windows 2000: Sets the underlining of menu access key letters. The pvParam parameter is a BOOL variable. Set pvParam to TRUE to always underline menu access keys, or FALSE to underline menu access keys only when the menu is activated from the keyboard.
SPI_SETKEYBOARDDELAY	Sets the keyboard repeat-delay setting. The uiParam parameter must specify 0, 1, 2, or 3, where zero sets the shortest delay (approximately 250 ms) and 3 sets the longest delay (approximately 1 second). The actual delay associated with each value may vary depending on the hardware.

SPI_SETKEYBOARDPREF	Windows 95/98, Windows 2000: Sets the keyboard preference. The uiParam parameter specifies TRUE if the user relies on the keyboard instead of the mouse, and wants applications to display keyboard interfaces that would otherwise be hidden; uiParam is FALSE otherwise.
SPI_SETKEYBOARDSPEED	Sets the keyboard repeat-speed setting. The uiParam parameter must specify a value in the range from 0 (approximately 2.5 repetitions per second) through 31 (approximately 30 repetitions per second). The actual repeat rates are hardware-dependent and may vary from a linear scale by as much as 20%. If uiParam is greater than 31, the parameter is set to 31.
SPI_SETLANGTOGGLE	Sets the hot key set for switching between input languages. The uiParam and pvParam parameters are not used. The value sets the shortcut keys in the keyboard property sheets by reading the registry again. The registry must be set before this flag is used. the path in the registry is \HKEY_CURRENT_USER\keyboard layout\toggle. Valid values are "1" = ALT+SHIFT, "2" = CTRL+SHIFT, and "3" = none.
SPI_SETMOUSE	Sets the two mouse threshold values and the mouse acceleration. The pvParam parameter must point to an array of three integers that specifies these values. See mouse_event for further information.
SPI_SETMOUSEBUTTONSWAP	Swaps or restores the meaning of the left and right mouse buttons. The uiParam parameter specifies TRUE to swap the meanings of the buttons, or FALSE to restore their original meanings.
SPI_SETMOUSEHOVERHEIGHT	Windows NT 4.0, Windows 98, Windows 2000: Sets the height, in pixels, of the rectangle within which the mouse pointer has to stay for TrackMouseEvent to generate a WM_MOUSEHOVER message. Set the uiParam parameter to the new height.
SPI_SETMOUSEHOVERTIME	Windows NT 4.0, Windows 98, Windows 2000: Sets the time, in milliseconds, that the mouse pointer has to stay in the hover rectangle for TrackMouseEvent to generate a WM_MOUSEHOVER message. This is used only if you pass HOVER_DEFAULT in the dwHoverTime parameter in the call to TrackMouseEvent. Set the uiParam parameter to the new time.

SPI_SETMOUSEHOVERWIDTH	Windows NT 4.0, Windows 98, Windows 2000: Sets the width, in pixels, of the rectangle within which the mouse pointer has to stay for TrackMouseEvent to generate a WM_MOUSEHOVER message. Set the uiParam parameter to the new width.
SPI_SETMOUSESPEED	Windows 98, Windows 2000: Sets the current mouse speed. The pvParam parameter is an integer between 1 (slowest) and 20 (fastest). A value of 10 is the default. This value is typically set using the mouse control panel application.
SPI_SETMOUSETRAILS	Windows 95/98 and Whistler: Enables or disables the Mouse Trails feature, which improves the visibility of mouse cursor movements by briefly showing a trail of cursors and quickly erasing them. To disable the feature, set the uiParam parameter to zero or 1. To enable the feature, set uiParam to a value greater than 1 to indicate the number of cursors drawn in the trail. Windows NT/2000: Not supported.
SPI_SETSNAPTODEFBUTTON	Windows NT 4.0, Windows 98, Windows 2000: Enables or disables the snap-to-default-button feature. If enabled, the mouse cursor automatically moves to the default button, such as OK or Apply, of a dialog box. Set the uiParam parameter to TRUE to enable the feature, or FALSE to disable it. Applications should use the ShowWindow function when displaying a dialog box so the dialog manager can position the mouse cursor.
SPI_SETWHEELSCROLLLINES	Windows NT 4.0, Windows 98, Windows 2000: Sets the number of lines to scroll when the mouse wheel is rotated. The number of lines is set from the uiParam parameter. The number of lines is the suggested number of lines to scroll when the mouse wheel is rolled without using modifier keys. If the number is 0, then no scrolling should occur. If the number of lines to scroll is greater than the number of lines viewable, and in particular if it is WHEEL_PAGESCROLL (#defined as UINT_MAX), the scroll operation should be interpreted as clicking once in the page down or page up regions of the scroll bar.

The following are the menu parameters.

Menu parameter	Meaning

SPI_GETMENUDROPALIGNMENT	Determines whether pop-up menus are left-aligned or right-aligned, relative to the corresponding menu-bar item. The *pvParam parameter must point to a BOOL variable that receives TRUE if left-aligned, or FALSE otherwise.*
SPI_GETMENUFADE	Windows 2000: Indicates whether menu fade animation is enabled. The pvParam parameter must point to a BOOL variable that receives TRUE when fade animation is enabled and FALSE when it is disabled. If fade animation is disabled, menus use slide animation. This flag is ignored unless menu animation is enabled, which you can do using the SPI_SETMENUANIMATION flag. For more information, see AnimateWindow.
SPI_GETMENUSHOWDELAY	Windows 95/98, Windows NT 4.0, Windows 2000: Indicates the time, in milliseconds, that the system waits before displaying a shortcut menu when the mouse cursor is over a submenu item. The pvParam parameter must point to a DWORD variable that receives the time of the delay.
SPI_SETMENUDROPALIGNMENT	Sets the alignment value of pop-up menus. The uiParam parameter specifies TRUE for right alignment, or FALSE for left alignment.
SPI_SETMENUFADE	Windows 2000: Enables or disables menu fade animation. Set pvParam to TRUE to enable the menu fade effect or FALSE to disable it. If fade animation is disabled, menus use slide animation. he The menu fade effect is possible only if the system has a color depth of more than 256 colors. This flag is ignored unless SPI_MENUANIMATION is also set. For more information, see AnimateWindow.
SPI_SETMENUSHOWDELAY	Windows 95/98, Windows NT 4.0, Windows 2000: Set uiParam to the time, in milliseconds, that the system waits before displaying a shortcut menu when the mouse cursor is over a submenu item.

The following are the power parameters.

Power parameter	Meaning

SPI_GETLOWPOWERACTIVE	Determines whether the low-power phase of screen saving is enabled. The *pvParam parameter must point to a BOOL variable that receives TRUE if enabled, or FALSE if disabled.* Windows 98: This flag is supported for 16-bit and 32-bit applications. Windows 95: This flag is supported for 16-bit applications only. Windows NT/2000: This flag is supported for 32-bit applications on Windows 2000 and later. It is not supported for 16-bit applications.
SPI_GETLOWPOWERTIMEOUT	Retrieves the time-out value for the low-power phase of screen saving. The pvParam parameter must point to an integer variable that receives the value. Windows 98: This flag is supported for 16-bit and 32-bit applications. Windows 95: This flag is supported for 16-bit applications only. Windows NT/2000: This flag is supported for 32-bit applications on Windows 2000 and later. It is not supported for 16-bit applications.
SPI_GETPOWEROFFACTIVE	Determines whether the power-off phase of screen saving is enabled. The pvParam parameter must point to a BOOL variable that receives TRUE if enabled, or FALSE if disabled. Windows 98: This flag is supported for 16-bit and 32-bit applications. Windows 95: This flag is supported for 16-bit applications only. Windows NT/2000: This flag is supported for 32-bit applications on Windows 2000 and later. It is not supported for 16-bit applications.

SPI_GETPOWEROFFTIMEOUT	Retrieves the time-out value for the power-off phase of screen saving. The pvParam parameter must point to an integer variable that receives the value. Windows 98: This flag is supported for 16-bit and 32-bit applications. Windows 95: This flag is supported for 16-bit applications only. Windows NT/2000: This flag is supported for 32-bit applications on Windows 2000 and later. It is not supported for 16-bit applications.
SPI_SETLOWPOWERACTIVE	Activates or deactivates the low-power phase of screen saving. Set uiParam to 1 to activate, or zero to deactivate. The pvParam parameter must be NULL. Windows 98: This flag is supported for 16-bit and 32-bit applications. Windows 95: This flag is supported for 16-bit applications only. Windows NT/2000: This flag is supported for 32-bit applications on Windows 2000 and later. It is not supported for 16-bit applications.
SPI_SETLOWPOWERTIMEOUT	Sets the time-out value, in seconds, for the low-power phase of screen saving. The uiParam parameter specifies the new value. The pvParam parameter must be NULL. Windows 98: This flag is supported for 16-bit and 32-bit applications. Windows 95: This flag is supported for 16-bit applications only. Windows NT/2000: This flag is supported for 32-bit applications on Windows 2000 and later. It is not supported for 16-bit applications.
SPI_SETPOWEROFFACTIVE	Activates or deactivates the power-off phase of screen saving. Set uiParam to 1 to activate, or zero to deactivate. The pvParam parameter must be NULL. Windows 98: This flag is supported for 16-bit and 32-bit applications. Windows 95: This flag is supported for 16-bit applications only. Windows NT/2000: This flag is supported for 32-bit applications on Windows 2000 and later. It is not supported for 16-bit applications.

SPI_SETPOWEROFFTIMEOUT	Sets the time-out value, in seconds, for the power-off phase of screen saving. The uiParam parameter specifies the new value. The pvParam parameter must be NULL. Windows 98: This flag is supported for 16-bit and 32-bit applications. Windows 95: This flag is supported for 16-bit applications only. Windows NT/2000: This flag is supported for 32-bit applications on Windows 2000 and later. It is not supported for 16-bit applications.

The following are the screen saver parameters.

Screen saver parameter	Meaning
SPI_GETSCREENSAVEACTIVE	Determines whether screen saving is enabled. The *pvParam parameter must point to a BOOL variable that receives TRUE if screen saving is enabled, or FALSE otherwise.*
SPI_GETSCREENSAVERRUNNING	Windows 98, Windows 2000: Determines whether a screen saver is currently running on the window station of the calling process. The pvParam parameter must point to a BOOL variable that receives TRUE if a screen saver is currently running, or FALSE otherwise. Note that only the interactive window station, "WinSta0", can have a screen saver running.
SPI_GETSCREENSAVETIMEOUT	Retrieves the screen saver time-out value, in seconds. The pvParam parameter must point to an integer variable that receives the value.
SPI_SETSCREENSAVEACTIVE	Sets the state of the screen saver. The uiParam parameter specifies TRUE to activate screen saving, or FALSE to deactivate it.
SPI_SETSCREENSAVERRUNNING	Windows 95/98: Used internally; applications should not use this flag.
SPI_SETSCREENSAVETIMEOUT	Sets the screen saver time-out value to the value of the uiParam parameter. This value is the amount of time, in seconds, that the system must be idle before the screen saver activates.

The following are the UI effects parameters. The SPI_SETUIEFFECTS value is used to enable or disable all the UI effect values at once. This table contains the complete list of UI effect values.

UI effects parameter	Meaning
SPI_GETCOMBOBOXANIMATION	Windows 98, Windows 2000: Indicates whether the slide-open effect for combo boxes is enabled. The *pvParam parameter must point to a BOOL variable that receives TRUE for enabled, or FALSE for disabled.*

SPI_GETCURSORSHADOW	Windows 2000: Indicates whether the cursor has a shadow around it. The pvParam parameter must point to a BOOL variable that receives TRUE if the shadow is enabled, FALSE if it is disabled. This effect appears only if the system has a color depth of more than 256 colors.
SPI_GETGRADIENTCAPTIONS	Windows 98, Windows 2000: Indicates whether the gradient effect for window title bars is enabled. The pvParam parameter must point to a BOOL variable that receives TRUE for enabled, or FALSE for disabled. For more information about the gradient effect, see the GetSysColor function.
SPI_GETHOTTRACKING	Windows 98, Windows 2000: Indicates whether hot tracking of user-interface elements, such as menu names on menu bars, is enabled. The pvParam parameter must point to a BOOL variable that receives TRUE for enabled, or FALSE for disabled. Hot tracking means that when the cursor moves over an item, it is highlighted but not selected. You can query this value to decide whether to use hot tracking in the user interface of your application.
SPI_GETLISTBOXSMOOTHSCROLLING	Windows 98, Windows 2000: Indicates whether the smooth-scrolling effect for list boxes is enabled. The pvParam parameter must point to a BOOL variable that receives TRUE for enabled, or FALSE for disabled.
SPI_GETMENUANIMATION	Windows 98, Windows 2000: Indicates whether the menu animation feature is enabled. This master switch must be on to enable menu animation effects. The pvParam parameter must point to a BOOL variable that receives TRUE if animation is enabled and FALSE if it is disabled. Windows 2000: If animation is enabled, SPI_GETMENUFADE indicates whether menus use fade or slide animation.
SPI_GETMENUUNDERLINES	Windows 98, Windows 2000: Same as SPI_GETKEYBOARDCUES.
SPI_GETSELECTIONFADE	Windows 2000: Indicates whether the selection fade effect is enabled. The pvParam parameter must point to a BOOL variable that receives TRUE if enabled or FALSE if disabled. The selection fade effect causes the menu item selected by the user to remain on the screen briefly while fading out after the menu is dismissed.

SPI_GETTOOLTIPANIMATION	Windows 2000: Indicates whether ToolTip animation is enabled. The pvParam parameter must point to a BOOL variable that receives TRUE if enabled or FALSE if disabled. If ToolTip animation is enabled, SPI_GETTOOLTIPFADE indicates whether ToolTips use fade or slide animation.
SPI_GETTOOLTIPFADE	Windows 2000: If SPI_SETTOOLTIPANIMATION is enabled, SPI_GETTOOLTIPFADE indicates whether ToolTip animation uses a fade effect or a slide effect. The pvParam parameter must point to a BOOL variable that receives TRUE for fade animation or FALSE for slide animation. For more information on slide and fade effects, see AnimateWindow.
SPI_GETUIEFFECTS	Windows 2000: Indicates whether all UI effects are disabled or not. The pvParam parameter must point to a BOOL variable that receives TRUE if UI effects are enabled, or FALSE if they are disabled. For the complete list of UI effects, see the Remarks section later in this topic.
SPI_SETCOMBOBOXANIMATION	Windows 98, Windows 2000: Enables or disables the slide-open effect for combo boxes. Set the pvParam parameter to TRUE to enable the gradient effect, or FALSE to disable it.
SPI_SETCURSORSHADOW	Windows 2000: Enables or disables a shadow around the cursor. The pvParam parameter is a BOOL variable. Set pvParam to TRUE to enable the shadow or FALSE to disable the shadow. This effect appears only if the system has a color depth of more than 256 colors.
SPI_SETGRADIENTCAPTIONS	Windows 98, Windows 2000: Enables or disables the gradient effect for window title bars. Set the pvParam parameter to TRUE to enable it, or FALSE to disable it. The gradient effect is possible only if the system has a color depth of more than 256 colors. For more information about the gradient effect, see the GetSysColor function.
SPI_SETHOTTRACKING	Windows 98, Windows 2000: Enables or disables hot tracking of user-interface elements such as menu names on menu bars. Set the pvParam parameter to TRUE to enable it, or FALSE to disable it. Hot-tracking means that when the cursor moves over an item, it is highlighted but not selected.
SPI_SETLISTBOXSMOOTHSCROLLING	Windows 98, Windows 2000: Enables or disables the smooth-scrolling effect for list boxes. Set the pvParam parameter to TRUE to enable the smooth-scrolling effect, or FALSE to disable it.

SPI_SETMENUANIMATION	Windows 98, Windows 2000: Enables or disables menu animation. This master switch must be on for any menu animation to occur. The pvParam parameter is a BOOL variable; set pvParam to TRUE to enable animation and FALSE to disable animation. Windows 2000: If animation is enabled, SPI_GETMENUFADE indicates whether menus use fade or slide animation.
SPI_SETMENUUNDERLINES	Windows 98, Windows 2000: Same as SPI_SETKEYBOARDCUES.
SPI_SETSELECTIONFADE	Windows 2000: Set pvParam to TRUE to enable the selection fade effect or FALSE to disable it. The selection fade effect causes the menu item selected by the user to remain on the screen briefly while fading out after the menu is dismissed. The selection fade effect is possible only if the system has a color depth of more than 256 colors.
SPI_SETTOOLTIPANIMATION	Windows 2000: Set pvParam to TRUE to enable ToolTip animation or FALSE to disable it. If enabled, you can use SPI_SETTOOLTIPFADE to specify fade or slide animation.
SPI_SETTOOLTIPFADE	Windows 2000: If the SPI_SETTOOLTIPANIMATION flag is enabled, use SPI_SETTOOLTIPFADE to indicate whether ToolTip animation uses a fade effect or a slide effect. Set pvParam to TRUE for fade animation or FALSE for slide animation. The tooltip fade effect is possible only if the system has a color depth of more than 256 colors. For more information on the slide and fade effects, see the AnimateWindow function.
SPI_SETUIEFFECTS	Windows 2000: Enables or disables UI effects. Set the pvParam parameter to TRUE to enable all UI effects or FALSE to disable all UI effects. For the complete list of UI effects, see the Remarks section later in this topic.

The following are the window parameters.

Window parameter	Meaning
SPI_GETACTIVEWINDOWTRACKING	Windows 98, Windows 2000: Indicates whether active window tracking (activating the window the mouse is on) is on or off. The *pvParam parameter must point to a BOOL variable that receives TRUE for on, or FALSE for off.*

SPI_GETACTIVEWNDTRKZORDER	Windows 98, Windows 2000: Indicates whether windows activated through active window tracking will be brought to the top. The pvParam parameter must point to a BOOL variable that receives TRUE for on, or FALSE for off.
SPI_GETACTIVEWNDTRKTIMEOUT	Windows 98, Windows 2000: Indicates the active window tracking delay, in milliseconds. The pvParam parameter must point to a DWORD variable that receives the time.
SPI_GETANIMATION	Retrieves the animation effects associated with user actions. The pvParam parameter must point to an ANIMATIONINFO structure that receives the information. Set the cbSize member of this structure and the uiParam parameter to sizeof(ANIMATIONINFO).
SPI_GETBORDER	Retrieves the border multiplier factor that determines the width of a window's sizing border. The pvParam parameter must point to an integer variable that receives this value.
SPI_GETCARETWIDTH	Windows 2000: Indicates the caret width in edit controls. The pvParam parameter must point to a DWORD that receives the width of the caret in pixels.
SPI_GETDRAGFULLWINDOWS	Determines whether dragging of full windows is enabled. The pvParam parameter must point to a BOOL variable that receives TRUE if enabled, or FALSE otherwise. Windows 95: This flag is supported only if Windows Plus! is installed. See SPI_GETWINDOWSEXTENSION.
SPI_GETFOREGROUNDFLASHCOUNT	Windows 98, Windows 2000: Indicates the number of times SetForegroundWindow will flash the taskbar button when rejecting a foreground switch request. The pvParam parameter must point to a DWORD variable that receives the value.
SPI_GETFOREGROUNDLOCKTIMEOUT	Windows 98, Windows 2000: Indicates the amount of time following user input, in milliseconds, during which the system will not allow applications to force themselves into the foreground. The pvParam parameter must point to a DWORD variable that receives the time.
SPI_GETMINIMIZEDMETRICS	Retrieves the metrics associated with minimized windows. The pvParam parameter must point to a MINIMIZEDMETRICS structure that receives the information. Set the cbSize member of this structure and the uiParam parameter to sizeof(MINIMIZEDMETRICS).

SPI_GETNONCLIENTMETRICS	Retrieves the metrics associated with the nonclient area of nonminimized windows. The pvParam parameter must point to a NONCLIENTMETRICS structure that receives the information. Set the cbSize member of this structure and the uiParam parameter to sizeof(NONCLIENTMETRICS).
SPI_GETSHOWIMEUI	Windows 98, Windows 2000: Indicates whether the IME status window is visible (on a per-user basis). The pvParam parameter must point to a BOOL variable that receives TRUE if the status window is visible, or FALSE if it is not.
SPI_SETACTIVEWINDOWTRACKING	Windows 98, Windows 2000: Sets active window tracking (activating the window the mouse is on) either on or off. Set pvParam to TRUE for on or FALSE for off.
SPI_SETACTIVEWNDTRKZORDER	Windows 98, Windows 2000: Indicates whether or not windows activated through active window tracking should be brought to the top. Set pvParam to TRUE for on or FALSE for off.
SPI_SETACTIVEWNDTRKTIMEOUT	Windows 98, Windows 2000: Sets the active window tracking delay. Set pvParam to the number of milliseconds to delay before activating the window under the mouse pointer.
SPI_SETANIMATION	Sets the animation effects associated with user actions. The pvParam parameter must point to an ANIMATIONINFO structure that contains the new parameters. Set the cbSize member of this structure and the uiParam parameter to sizeof(ANIMATIONINFO).
SPI_SETBORDER	Sets the border multiplier factor that determines the width of a window's sizing border. The uiParam parameter specifies the new value.
SPI_SETCARETWIDTH	Windows 2000: Sets the caret width in edit controls. Set pvParam to the desired width, in pixels. The default and minimum value is 1.
SPI_SETDRAGFULLWINDOWS	Sets dragging of full windows either on or off. The uiParam parameter specifies TRUE for on, or FALSE for off. Windows 95: This flag is supported only if Windows Plus! is installed. See SPI_GETWINDOWSEXTENSION.
SPI_SETDRAGHEIGHT	Sets the height, in pixels, of the rectangle used to detect the start of a drag operation. Set uiParam to the new value. To retrieve the drag height, call GetSystemMetrics with the SM_CYDRAG flag.

SPI_SETDRAGWIDTH	Sets the width, in pixels, of the rectangle used to detect the start of a drag operation. Set uiParam to the new value. To retrieve the drag width, call GetSystemMetrics with the SM_CXDRAG flag.
SPI_SETFOREGROUNDFLASHCOUNT	Windows 98, Windows 2000: Sets the number of times SetForegroundWindow will flash the taskbar button when rejecting a foreground switch request. Set pvParam to the number of times to flash.
SPI_SETFOREGROUNDLOCKTIMEOUT	Windows 98, Windows 2000: Sets the amount of time following user input, in milliseconds, during which the system will not allow applications to force themselves into the foreground. Set pvParam to the new timeout value.
SPI_SETMINIMIZEDMETRICS	Sets the metrics associated with minimized windows. The pvParam parameter must point to a MINIMIZEDMETRICS structure that contains the new parameters. Set the cbSize member of this structure and the uiParam parameter to sizeof(MINIMIZEDMETRICS).
SPI_SETNONCLIENTMETRICS	Sets the metrics associated with the nonclient area of nonminimized windows. The pvParam parameter must point to a NONCLIENTMETRICS structure that contains the new parameters. Set the cbSize member of this structure and the uiParam parameter to sizeof(NONCLIENTMETRICS).
SPI_SETSHOWIMEUI	Windows 98, Windows 2000: Sets whether the IME status window is visible or not on a per-user basis. The uiParam parameter specifies TRUE for on or FALSE for off.

The following parameters are specific to Windows 95 and Windows 98.

Windows 95/98 parameter	Meaning
SPI_GETWINDOWSEXTENSION	Windows 95: Indicates whether the Windows extension, Windows Plus!, is installed. Set the *uiParam parameter to 1. The pvParam parameter is not used. The function returns TRUE if the extension is installed, or FALSE if it is not.*
SPI_SETPENWINDOWS	Windows 95/98: Specifies that pen windows is being loaded or unloaded. The uiParam parameter is TRUE when loading and FALSE when unloading pen windows. The pvParam parameter is NULL.

uiParam

[in] Depends on the system parameter being queried or set. For more information about system-wide parameters, see the uiAction parameter. If not otherwise indicated, you must specify zero for this parameter.

pvParam

[in/out] Depends on the system parameter being queried or set. For more information about system-wide parameters, see the uiAction parameter. If not otherwise indicated, you must specify NULL for this parame-

ter.

fWinIni

[in] If a system parameter is being set, specifies whether the user profile is to be updated, and if so, whether the WM_SETTINGCHANGE message is to be broadcast to all top-level windows to notify them of the change.

This parameter can be zero if you don't want to update the user profile or broadcast the WM_SETTINGCHANGE message, or it can be one or more of the following values.

Value	Action
SPIF_UPDATEINIFILE	Writes the new system-wide parameter setting to the user profile.
SPIF_SENDCHANGE	Broadcasts the WM_SETTINGCHANGE message after updating the user profile.
SPIF_SENDWININICHANGE	Same as SPIF_SENDCHANGE.

Return Values

If the function succeeds, the return value is a nonzero value.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

This function is intended for use with applications that allow the user to customize the environment.

A keyboard layout name should be derived from the hexadecimal value of the language identifier corresponding to the layout. For example, U.S. English has a language identifier of 0x0409, so the primary U.S. English layout is named "00000409." Variants of U.S. English layout, such as the Dvorak layout, are named "00010409," "00020409" and so on. For a list of the primary language identifiers and sublanguage identifiers that make up a language identifier, see the MAKELANGID macro.

Windows Me and Whistler: During the time that the primary button is held down to activate the Mouse Click-Lock feature, the user can move the mouse. Once the primary button is locked down, releasing the primary button does not result in a WM_LBUTTONUP message. Thus, it will appear to an application that the primary button is still down. Any subsequent button message releases the primary button, sending a WM_LBUTTONUP message to the application, thus the button can be unlocked programmatically or through the user clicking any button.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.415 TabbedTextOut

The TabbedTextOut function writes a character string at a specified location, expanding tabs to the values specified in an array of tab-stop positions. Text is written in the currently selected font, background color, and text color.

```
TabbedTextOut: procedure
(
        hDC                 :dword;
        X                   :dword;
        Y                   :dword;
        lpString            :string;
        nCount              :dword;
        nTabPositions       :dword;
    var lpnTabStopPositions :dword;
```

```
        nTabOrigin          :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__TabbedTextOutA@32" );
```

Parameters

hDC

[in] Handle to the device context.

X

[in] Specifies the x-coordinate of the starting point of the string, in logical units.

Y

[in] Specifies the y-coordinate of the starting point of the string, in logical units.

lpString

[in] Pointer to the character string to draw. The string does not need to be zero-terminated, since nCount specifies the length of the string.

nCount

[in] Specifies the length of the string pointed to by lpString. For the ANSI function it is a BYTE count and for the Unicode function it is a WORD count. Note that for the ANSI function, characters in SBCS code pages take one byte each, while most characters in DBCS code pages take two bytes; for the Unicode function, most currently defined Unicode characters (those in the Basic Multilingual Plane (BMP)) are one WORD while Unicode surrogates are two WORDs.

Windows 95/98: This value may not exceed 8192.

nTabPositions

[in] Specifies the number of values in the array of tab-stop positions.

lpnTabStopPositions

[in] Pointer to an array containing the tab-stop positions, in logical units. The tab stops must be sorted in increasing order; the smallest x-value should be the first item in the array.

Windows 95: A tab stop can be specified as a negative value, which causes text to be right-aligned on the tab stop rather than left-aligned.

nTabOrigin

[in] Specifies the x-coordinate of the starting position from which tabs are expanded, in logical units.

Return Values

If the function succeeds, the return value is the dimensions, in logical units, of the string. The height is in the high-order word and the width is in the low-order word.

If the function fails, the return value is zero.

Windows NT/ 2000: To get extended error information, call GetLastError.

Remarks

If the nTabPositions parameter is zero and the lpnTabStopPositions parameter is NULL, tabs are expanded to eight times the average character width.

If nTabPositions is 1, the tab stops are separated by the distance specified by the first value in the lpnTabStopPositions array.

If the lpnTabStopPositions array contains more than one value, a tab stop is set for each value in the array, up to

the number specified by nTabPositions.

The nTabOrigin parameter allows an application to call the TabbedTextOut function several times for a single line. If the application calls TabbedTextOut more than once with the nTabOrigin set to the same value each time, the function expands all tabs relative to the position specified by nTabOrigin.

By default, the current position is not used or updated by the TabbedTextOut function. If an application needs to update the current position when it calls TabbedTextOut, the application can call the SetTextAlign function with the wFlags parameter set to TA_UPDATECP. When this flag is set, the system ignores the X and Y parameters on subsequent calls to the TabbedTextOut function, using the current position instead.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.416 TileWindows

The TileWindows function tiles the specified child windows of the specified parent window.

```
TileWindows: procedure
(
        hwndParent  :dword;
        wHow        :dword;
    var lpRect      :RECT;
        cKids       :dword;
    var lpKids      :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__TileWindows@20" );
```

Parameters

hwndParent

[in] Handle to the parent window. If this parameter is NULL, the desktop window is assumed.

wHow

[in] Specifies tiling flags. This parameter can be one of the following values—optionally combined with MDITILE_SKIPDISABLED to prevent disabled MDI-child windows from being tiled.

Value	Meaning
MDITILE_HORIZONTAL	Tiles windows horizontally.
MDITILE_VERTICAL	Tiles windows vertically.

lpRect

[in] Pointer to a RECT structure that specifies the rectangular area, in client coordinates, within which the windows are arranged. If this parameter is NULL, the client area of the parent window is used.

cKids

[in] Specifies the number of elements in the array specified by the lpKids parameter. This parameter is ignored if lpKids is NULL.

lpKids

[in] Pointer to an array of handles to the child windows to arrange. If this parameter is NULL, all child windows of the specified parent window (or of the desktop window) are arranged.

Return Values

If the function succeeds, the return value is the number of windows arranged.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

Calling TileWindows causes all maximized windows to be restored to their previous size.

Requirements

Windows NT/2000: Requires Windows NT 4.0 or later.

Windows 95/98: Requires Windows 95 or later.

3.417 ToAscii

The ToAscii function translates the specified virtual-key code and keyboard state to the corresponding character or characters. The function translates the code using the input language and physical keyboard layout identified by the keyboard layout handle.

To specify a handle to the keyboard layout to use to translate the specified code, use the ToAsciiEx function.

```
ToAscii: procedure
(
        uVirtKey    :dword;
        uScanCode   :dword;
    var kpKeyState  :var;
    var lpChar      :var;
        uFlags      :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__ToAscii@20" );
```

Parameters

uVirtKey

[in] Specifies the virtual-key code to be translated.

uScanCode

[in] Specifies the hardware scan code of the key to be translated. The high-order bit of this value is set if the key is up (not pressed).

lpKeyState

[in] Pointer to a 256-byte array that contains the current keyboard state. Each element (byte) in the array contains the state of one key. If the high-order bit of a byte is set, the key is down (pressed).

The low bit, if set, indicates that the key is toggled on. In this function, only the toggle bit of the CAPS LOCK key is relevant. The toggle state of the NUM LOCK and SCROLL LOCK keys is ignored.

lpChar

[out] Pointer to the buffer that receives the translated character or characters.

uFlags

[in] Specifies whether a menu is active. This parameter must be 1 if a menu is active, or 0 otherwise.

Return Values

If the specified key is a dead key, the return value is negative. Otherwise, it is one of the following values.

Value	Meaning
0	The specified virtual key has no translation for the current state of the keyboard.
1	One character was copied to the buffer.
2	Two characters were copied to the buffer. This usually happens when a dead-key character (accent or diacritic) stored in the keyboard layout cannot be composed with the specified virtual key to form a single character.

Remarks

The parameters supplied to the ToAscii function might not be sufficient to translate the virtual-key code, because a previous dead key is stored in the keyboard layout.

Typically, ToAscii performs the translation based on the virtual-key code. In some cases, however, bit 15 of the uScanCode parameter may be used to distinguish between a key press and a key release. The scan code is used for translating ALT+number key combinations.

Although NUM LOCK is a toggle key that affects keyboard behavior, ToAscii ignores the toggle setting (the low bit) of lpKeyState (VK_NUMLOCK, because the uVirtKey parameter alone is sufficient to distinguish the cursor movement keys (VK_HOME, VK_INSERT, and so on) from the numeric keys (VK_DECIMAL, VK_NUMPAD0 - VK_NUMPAD9).

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.418 ToAsciiEx

The ToAsciiEx function translates the specified virtual-key code and keyboard state to the corresponding character or characters. The function translates the code using the input language and physical keyboard layout identified by the input locale identifier.

```
ToAsciiEx: procedure
(
        uVirtKey    :dword;
        uScanCode   :dword;
    var kpKeyState  :var;
    var lpChar      :var;
        uFlags      :dword;
        dwhkl       :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__ToAsciiEx@24" );
```

Parameters

uVirtKey

[in] Specifies the virtual-key code to be translated.

uScanCode

[in] Specifies the hardware scan code of the key to be translated. The high-order bit of this value is set if the

key is up (not pressed).

lpKeyState

[in] Pointer to a 256-byte array that contains the current keyboard state. Each element (byte) in the array contains the state of one key. If the high-order bit of a byte is set, the key is down (pressed).

The low bit, if set, indicates that the key is toggled on. In this function, only the toggle bit of the CAPS LOCK key is relevant. The toggle state of the NUM LOCK and SCOLL LOCK keys is ignored.

lpChar

[out] Pointer to the buffer that receives the translated character or characters.

uFlags

[in] Specifies whether a menu is active. This parameter must be 1 if a menu is active, zero otherwise.

dwhkl

[in] Input locale identifier to use to translate the code. This parameter can be any input locale identifier previously returned by the LoadKeyboardLayout function.

Return Values

If the specified key is a dead key, the return value is negative. Otherwise, it is one of the following values.

Value	Meaning
0	The specified virtual key has no translation for the current state of the keyboard.
1	One character was copied to the buffer.
2	Two characters were copied to the buffer. This usually happens when a dead-key character (accent or diacritic) stored in the keyboard layout cannot be composed with the specified virtual key to form a single character.

Remarks

The input locale identifier is a broader concept than a keyboard layout, since it can also encompass a speech-to-text converter, an IME, or any other form of input.

The parameters supplied to the ToAsciiEx function might not be sufficient to translate the virtual-key code, because a previous dead key is stored in the keyboard layout.

Typically, ToAsciiEx performs the translation based on the virtual-key code. In some cases, however, bit 15 of the uScanCode parameter may be used to distinguish between a key press and a key release. The scan code is used for translating ALT+number key combinations.

Although NUM LOCK is a toggle key that affects keyboard behavior, ToAsciiEx ignores the toggle setting (the low bit) of lpKeyState (VK_NUMLOCK, because the uVirtKey parameter alone is sufficient to distinguish the cursor movement keys (VK_HOME, VK_INSERT, and so on) from the numeric keys (VK_DECIMAL, VK_NUMPAD0 - VK_NUMPAD9).

Requirements

Windows NT/2000: Requires Windows NT 4.0 or later.

Windows 95/98: Requires Windows 95 or later.

3.419 ToUnicode

The ToUnicode function translates the specified virtual-key code and keyboard state to the corresponding Unicode character or characters.

To specify a handle to the keyboard layout to use to translate the specified code, use the ToUnicodeEx function.

```
ToUnicode: procedure
(
        wVirtKey    :dword;
        wScanCode   :dword;
    var lpKeyState  :var;
    var pwszBuff    :var;
        cchBuff     :dword;
        wFlags      :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__ToUnicode@24" );
```

Parameters

wVirtKey

[in] Specifies the virtual-key code to be translated.

wScanCode

[in] Specifies the hardware scan code of the key to be translated. The high-order bit of this value is set if the key is up.

lpKeyState

[in] Pointer to a 256-byte array that contains the current keyboard state. Each element (byte) in the array contains the state of one key. If the high-order bit of a byte is set, the key is down.

pwszBuff

[out] Pointer to the buffer that receives the translated Unicode character or characters. Note, however, that this buffer may be returned without being NULL-terminated even though the variable name suggests that it is NULL-terminated.

cchBuff

[in] Specifies the size, in wide characters, of the buffer pointed to by the pwszBuff parameter.

wFlags

[in] Specifies the behavior of the function. If bit 0 is set, a menu is active. Bits 1 through 31 are reserved.

Return Values

The function returns one of the following values.

Value	Meaning
– 1	The specified virtual key is a dead-key character (accent or diacritic). This value is returned regardless of the keyboard layout, even if several characters have been typed and are stored in the keyboard state. If possible, even with Unicode keyboard layouts, the function has written a spacing version of the dead-key character to the buffer specified by pwszBuffer. For example, the function writes the character SPACING ACUTE (0x00B4), rather than the character NON_SPACING ACUTE (0x0301).
0	The specified virtual key has no translation for the current state of the keyboard. Nothing was written to the buffer specified by pwszBuffer.
1	One character was written to the buffer specified by pwszBuffer.

2 or more — Two or more characters were written to the buffer specified by pwszBuff. The most common cause for this is that a dead-key character (accent or diacritic) stored in the keyboard layout could not be combined with the specified virtual key to form a single character. However, the buffer may contain more characters than the return value specifies. When this happens, any extra characters are invalid and should be ignored.

Remarks

The parameters supplied to the ToUnicode function might not be sufficient to translate the virtual-key code because a previous dead key is stored in the keyboard layout.

Typically, ToUnicode performs the translation based on the virtual-key code. In some cases, however, bit 15 of the wScanCode parameter can be used to distinguish between a key press and a key release.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Unsupported.

3.420 ToUnicodeEx

The ToUnicodeEx function translates the specified virtual-key code and keyboard state to the corresponding Unicode character or characters.

```
ToUnicodeEx: procedure
(
        wVirtKey    :dword;
        wScanCode   :dword;
    var lpKeyState  :var;
    var pwszBuff    :var;
        cchBuff     :dword;
        wFlags      :dword;
        dwhkl       :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__ToUnicodeEx@28" );
```

Parameters

wVirtKey

[in] Specifies the virtual-key code to be translated.

wScanCode

[in] Specifies the hardware scan code of the key to be translated. The high-order bit of this value is set if the key is up.

lpKeyState

[in] Pointer to a 256-byte array that contains the current keyboard state. Each element (byte) in the array contains the state of one key. If the high-order bit of a byte is set, the key is down.

pwszBuff

[out] Pointer to the buffer that receives the translated Unicode character or characters. Note, however, that this buffer may be returned without being NULL-terminated even though the variable name suggests that it is NULL-terminated.

cchBuff

[in] Specifies the size, in wide characters, of the buffer pointed to by the pwszBuff parameter.

wFlags

[in] Specifies the behavior of the function. If bit 0 is set, a menu is active. Bits 1 through 31 are reserved.

dwhkl

[in] Input locale identifier used to translate the specified code. This parameter can be any input locale identifier previously returned by the LoadKeyboardLayout function.

Return Values

The function returns one of the following values.

Value	Meaning
–1	The specified virtual key is a dead-key character (accent or diacritic). This value is returned regardless of the keyboard layout, even if several characters have been typed and are stored in the keyboard state. If possible, even with Unicode keyboard layouts, the function has written a spacing version of the dead-key character to the buffer specified by pwszBuffer. For example, the function writes the character SPACING ACUTE (0x00B4), rather than the character NON_SPACING ACUTE (0x0301).
0	The specified virtual key has no translation for the current state of the keyboard. Nothing was written to the buffer specified by pwszBuffer.
1	One character was written to the buffer specified by pwszBuffer.
2 or more	Two or more characters were written to the buffer specified by pwszBuff. The most common cause for this is that a dead-key character (accent or diacritic) stored in the keyboard layout could not be combined with the specified virtual key to form a single character. However, the buffer may contain more characters than the return value specifies. When this happens, any extra characters are invalid and should be ignored.

Remarks

The input locale identifier is a broader concept than a keyboard layout, since it can also encompass a speech-to-text converter, an IME, or any other form of input.

The parameters supplied to the ToUnicodeEx function might not be sufficient to translate the virtual-key code because a previous dead key is stored in the keyboard layout.

Typically, ToUnicodeEx performs the translation based on the virtual-key code. In some cases, however, bit 15 of the wScanCode parameter can be used to distinguish between a key press and a key release.

Requirements

Windows NT/2000: Requires Windows NT 4.0 or later.

Windows 95/98: Unsupported.

3.421 TrackMouseEvent

The TrackMouseEvent function posts messages when the mouse pointer leaves a window or hovers over a window for a specified amount of time.

```
TrackMouseEvent: procedure
(
```

```
    var lpEventTrack    :TRACKMOUSEEVENT
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__TrackMouseEvent@4" );
```

Parameters

lpEventTrack

[in/out] Pointer to a TRACKMOUSEEVENT structure that contains tracking information.

Return Values

If the function succeeds, the return value is nonzero .

If the function fails, return value is zero. To get extended error information, call GetLastError.

The function can post the following messages.

Message	Meaning
WM_NCMOUSEHOVER	Windows 98, Windows 2000: The same meaning as WM_MOUSEHOVER except this is for the nonclient area of the window.
WM_NCMOUSELEAVE	Windows 98, Windows 2000: The same meaning as WM_MOUSELEAVE except this is for the nonclient area of the window.
WM_MOUSEHOVER	The mouse hovered over the client area of the window for the period of time specified in a prior call to . Hover tracking stops when this message is generated. The application must call again if it requires further tracking of mouse hover behavior.
WM_MOUSELEAVE	The mouse left the client area of the window specified in a prior call to . All tracking requested by is canceled when this message is generated. The application must call when the mouse reenters its window if it requires further tracking of mouse hover behavior.

Remarks

The mouse pointer is considered to be hovering when it stays within a specified rectangle for a specified period of time. Call SystemParametersInfo and use the values SPI_GETMOUSEHOVERWIDTH, SPI_GETMOUSEHOVERHEIGHT, and SPI_GETMOUSEHOVERTIME to retrieve the size of the rectangle and the time.

Note The _TrackMouseEvent function calls TrackMouseEvent if it exists, otherwise _TrackMouseEvent emulates TrackMouseEvent. The _TrackMouseEvent function is in commctrl.h and is exported by COMCTRL32.DLL.

Requirements

Windows NT/2000: Requires Windows NT 4.0 or later.

Windows 95/98: Requires Windows 98 or later.

3.422 TrackPopupMenu

The TrackPopupMenu function displays a shortcut menu at the specified location and tracks the selection of

items on the menu. The shortcut menu can appear anywhere on the screen.

To specify an area of the screen the menu should not overlap, use the TrackPopupMenuEx function.

```
TrackPopupMenu: procedure
(
        hMenu       :dword;
        uFlags      :dword;
        x           :dword;
        y           :dword;
        nReserved   :dword;
        hWnd        :dword;
    var prcRect     :RECT
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__TrackPopupMenu@28" );
```

Parameters

hMenu

[in] Handle to the shortcut menu to be displayed. The handle can be obtained by calling CreatePopupMenu to create a new shortcut menu, or by calling GetSubMenu to retrieve a handle to a submenu associated with an existing menu item.

uFlags

[in] Use zero of more of these flags to specify function options.

Use one of the following flags to specify how the function positions the shortcut menu horizontally.

Value	Meaning
TPM_CENTERALIGN	If this flag is set, the function centers the shortcut menu horizontally relative to the coordinate specified by the *x parameter.*
TPM_LEFTALIGN	If this flag is set, the function positions the shortcut menu so that its left side is aligned with the coordinate specified by the x parameter.
TPM_RIGHTALIGN	Positions the shortcut menu so that its right side is aligned with the coordinate specified by the x parameter.

Use one of the following flags to specify how the function positions the shortcut menu vertically.

Value	Meaning
TPM_BOTTOMALIGN	If this flag is set, the function positions the shortcut menu so that its bottom side is aligned with the coordinate specified by the y parameter.
TPM_TOPALIGN	If this flag is set, the function positions the shortcut menu so that its top side is aligned with the coordinate specified by the y parameter.
TPM_VCENTERALIGN	If this flag is set, the function centers the shortcut menu vertically relative to the coordinate specified by the y parameter.

Use the following flags to determine the user selection without having to set up a parent window for the menu.

Value	Meaning
TPM_NONOTIFY	If this flag is set, the function does not send notification messages when the user clicks on a menu item.
TPM_RETURNCMD	If this flag is set, the function returns the menu item identifier of the user's selection in the return value.

Use one of the following flags to specify which mouse button the shortcut menu tracks.

Value	Meaning
TPM_LEFTBUTTON	If this flag is set, the user can select menu items with only the left mouse button.
TPM_RIGHTBUTTON	If this flag is set, the user can select menu items with both the left and right mouse buttons.

Windows 98, Windows 2000: Use one of the following flags to modify the animation of a menu.

Value	Meaning
TPM_HORNEGANIMATION	Animates the menu from right to left.
TPM_HORPOSANIMATION	Animates the menu from left to right.
TPM_NOANIMATION	Displays menu without animation.
TPM_VERNEGANIMATION	Animates the menu from bottom to top.
TPM_VERPOSANIMATION	Animates the menu from top to bottom.

For any animation to occur, the SystemParametersInfo function must set SPI_SETMENUANIMATION. Also, all the TPM_*ANIMATION flags, except TPM_NOANIMATION, are ignored if menu fade animation is on, See the SPI_GETMENUFADE flag in SystemParametersInfo.

Windows 98, Windows 2000: Use the TPM_RECURSE flag to display a menu when another menu is already displayed. This is intended to support context menus within a menu.

x

[in] Specifies the horizontal location of the shortcut menu, in screen coordinates.

y

[in] Specifies the vertical location of the shortcut menu, in screen coordinates.

nReserved

Reserved; must be zero.

hWnd

[in] Handle to the window that owns the shortcut menu. This window receives all messages from the menu. The window does not receive a WM_COMMAND message from the menu until the function returns.

If you specify TPM_NONOTIFY in the uFlags parameter, the function does not send messages to the window identified by hWnd. However, you must still pass a window handle in hWnd. It can be any window handle from your application.

prcRect

Ignored.

Return Values

If you specify TPM_RETURNCMD in the uFlags parameter, the return value is the menu-item identifier of the item that the user selected. If the user cancels the menu without making a selection, or if an error occurs, then the return value is zero.

If you do not specify TPM_RETURNCMD in the uFlags parameter, the return value is nonzero if the function succeeds and zero if it fails. To get extended error information, call GetLastError.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.423 TrackPopupMenuEx

The TrackPopupMenuEx function displays a shortcut menu at the specified location and tracks the selection of items on the shortcut menu. The shortcut menu can appear anywhere on the screen.

```
TrackPopupMenuEx: procedure
(
        hmenu       :dword;
        fuFlags     :dword;
        x           :dword;
        y           :dword;
        hwnd        :dword;
    var lptpm       :TPMPARAMS
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__TrackPopupMenuEx@24" );
```

Parameters

hmenu

[in] Handle to the shortcut menu to be displayed. This handle can be obtained by calling the CreatePopupMenu function to create a new shortcut menu or by calling the GetSubMenu function to retrieve a handle to a submenu associated with an existing menu item.

fuFlags

[in] Specifies function options.

Use one of the following flags to specify how the function positions the shortcut menu horizontally.

Value	Meaning
TPM_CENTERALIGN	If this flag is set, the function centers the shortcut menu horizontally relative to the coordinate specified by the *x parameter.*
TPM_LEFTALIGN	If this flag is set, the function positions the shortcut menu so that its left side is aligned with the coordinate specified by the x parameter.
TPM_RIGHTALIGN	Positions the shortcut menu so that its right side is aligned with the coordinate specified by the x parameter.

Use one of the following flags to specify how the function positions the shortcut menu vertically.

Value	Meaning
TPM_BOTTOMALIGN	If this flag is set, the function positions the shortcut menu so that its bottom side is aligned with the coordinate specified by the y parameter.
TPM_TOPALIGN	If this flag is set, the function positions the shortcut menu so that its top side is aligned with the coordinate specified by the y parameter.
TPM_VCENTERALIGN	If this flag is set, the function centers the shortcut menu vertically relative to the coordinate specified by the y parameter.

Use the following flags to determine the user selection without having to set up a parent window for the menu.

Value	Meaning
TPM_NONOTIFY	If this flag is set, the function does not send notification messages when the user clicks on a menu item.
TPM_RETURNCMD	If this flag is set, the function returns the menu item identifier of the user's selection in the return value.

Use one of the following flags to specify which mouse button the shortcut menu tracks.

Value	Meaning
TPM_LEFTBUTTON	If this flag is set, the user can select menu items with only the left mouse button.
TPM_RIGHTBUTTON	If this flag is set, the user can select menu items with both the left and right mouse buttons.

Windows 98, Windows 2000: Use one of the following flags to modify the animation of a menu.

Value	Meaning
TPM_HORNEGANIMATION	Animates the menu from left to right.
TPM_HORPOSANIMATION	Animates the menu from right to left.
TPM_NOANIMATION	Displays menu without animation.
TPM_VERNEGANIMATION	Animates the menu from bottom to top.
TPM_VERPOSANIMATION	Animates the menu from top to bottom.

For any animation to occur, the SystemParametersInfo function must set SPI_SETMENUANIMATION. Also, all the TPM_*ANIMATION flags, except TPM_NOANIMATION, are ignored if menu fade animation is on, See the SPI_GETMENUFADE flag in SystemParametersInfo.
Windows 98, Windows 2000: Use the TPM_RECURSE flag to display a menu when another menu is already displayed. This is intended to support context menus within a menu.

Use one of the following flags to specify whether to accommodate horizontal or vertical alignment.

Value	Meaning
TPM_HORIZONTAL	If the menu cannot be shown at the specified location without overlapping the excluded rectangle, the system tries to accommodate the requested horizontal alignment before the requested vertical alignment.
TPM_VERTICAL	If the menu cannot be shown at the specified location without overlapping the excluded rectangle, the system tries to accommodate the requested vertical alignment before the requested horizontal alignment.

The excluded rectangle is a portion of the screen that the menu should not overlap; it is specified by lptpm.

x

[in] Horizontal location of the shortcut menu, in screen coordinates.

y

[in] Vertical location of the shortcut menu, in screen coordinates.

hwnd

[in] Handle to the window that owns the shortcut menu. This window receives all messages from the menu. The window does not receive a WM_COMMAND message from the menu until the function returns.

If you specify TPM_NONOTIFY in the fuFlags parameter, the function does not send messages to the window identified by hwnd. However, you still have to pass a window handle in hwnd. It can be any window handle from your application.

lptpm

[in] Pointer to a TPMPARAMS structure that specifies an area of the screen the menu should not overlap. This parameter can be NULL.

Return Values

If you specify TPM_RETURNCMD in the fuFlags parameter, the return value is the menu-item identifier of the item that the user selected. If the user cancels the menu without making a selection, or if an error occurs, then the return value is zero.

If you do not specify TPM_RETURNCMD in the fuFlags parameter, the return value is nonzero if the function

succeeds and zero if it fails. To get extended error information, call GetLastError.

Requirements

Windows NT/2000: Requires Windows NT 4.0 or later.

Windows 95/98: Requires Windows 95 or later.

3.424 TranslateAccelerator

The TranslateAccelerator function processes accelerator keys for menu commands. The function translates a WM_KEYDOWN or WM_SYSKEYDOWN message to a WM_COMMAND or WM_SYSCOMMAND message (if there is an entry for the key in the specified accelerator table) and then sends the WM_COMMAND or WM_SYSCOMMAND message directly to the appropriate window procedure. TranslateAccelerator does not return until the window procedure has processed the message.

```
TranslateAccelerator: procedure
(
        hWnd        :dword;
        hAccTable   :dword;
    var lpMsg       :MSG
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__TranslateAcceleratorA@12" );

TranslateAcceleratorW: procedure
(
        hWnd        :dword;
        hAccTable   :dword;
    var lpMsg       :MSG
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__TranslateAccelerator@12" );
```

Parameters

hWnd

[in] Handle to the window whose messages are to be translated.

hAccTable

[in] Handle to the accelerator table. The accelerator table must have been loaded by a call to the LoadAccelerators function or created by a call to the CreateAcceleratorTable function.

lpMsg

[in] Pointer to an MSG structure that contains message information retrieved from the calling thread's message queue using the GetMessage or PeekMessage function.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

To differentiate the message that this function sends from messages sent by menus or controls, the high-order

word of the wParam parameter of the WM_COMMAND or WM_SYSCOMMAND message contains the value 1.

Accelerator key combinations used to select items from the window menu are translated into WM_SYSCOMMAND messages; all other accelerator key combinations are translated into WM_COMMAND messages.

When TranslateAccelerator returns a nonzero value and the message is translated, the application should not use the TranslateMessage function to process the message again.

An accelerator need not correspond to a menu command.

If the accelerator command corresponds to a menu item, the application is sent WM_INITMENU and WM_INITMENUPOPUP messages, as if the user were trying to display the menu. However, these messages are not sent if any of the following conditions exist:

- The window is disabled.
- The accelerator key combination does not correspond to an item on the window menu and the window is minimized.
- A mouse capture is in effect. For information about mouse capture, see the SetCapture function.

If the specified window is the active window and no window has the keyboard focus (which is generally the case if the window is minimized), TranslateAccelerator translates WM_SYSKEYUP and WM_SYSKEYDOWN messages instead of WM_KEYUP and WM_KEYDOWN messages.

If an accelerator keystroke occurs that corresponds to a menu item when the window that owns the menu is minimized, TranslateAccelerator does not send a WM_COMMAND message. However, if an accelerator keystroke occurs that does not match any of the items in the window's menu or in the window menu, the function sends a WM_COMMAND message, even if the window is minimized.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.425 TranslateMDISysAccel

The TranslateMDISysAccel function processes accelerator keystrokes for window menu commands of the multiple document interface (MDI) child windows associated with the specified MDI client window. The function translates WM_KEYUP and WM_KEYDOWN messages to WM_SYSCOMMAND messages and sends them to the appropriate MDI child windows.

```
TranslateMDISysAccel: procedure
(
        hWndClient  :dword;
    var lpMsg       :MSG
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__TranslateMDISysAccel@8" );
```

Parameters

hWndClient

[in] Handle to the MDI client window.

lpMsg

[in] Pointer to a message retrieved by using the GetMessage or PeekMessage function. The message must be an MSG structure and contain message information from the application's message queue.

Return Values

If the message is translated into a system command, the return value is nonzero.

If the message is not translated into a system command, the return value is zero.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.426 TranslateMessage

The TranslateMessage function translates virtual-key messages into character messages. The character messages are posted to the calling thread's message queue, to be read the next time the thread calls the GetMessage or PeekMessage function.

```
TranslateMessage: procedure
(
    var lpMsg   :MSG
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__TranslateMessage@4" );
```

Parameters

lpMsg

[in] Pointer to an MSG structure that contains message information retrieved from the calling thread's message queue by using the GetMessage or PeekMessage function.

Return Values

If the message is translated (that is, a character message is posted to the thread's message queue), the return value is nonzero.

If the message is WM_KEYDOWN, WM_KEYUP, WM_SYSKEYDOWN, or WM_SYSKEYUP, the return value is nonzero, regardless of the translation.

If the message is not translated (that is, a character message is not posted to the thread's message queue), the return value is zero.

Remarks

The TranslateMessage function does not modify the message pointed to by the lpMsg parameter.

WM_KEYDOWN and WM_KEYUP combinations produce a WM_CHAR or WM_DEADCHAR message. WM_SYSKEYDOWN and WM_SYSKEYUP combinations produce a WM_SYSCHAR or WM_SYSDEADCHAR message.

TranslateMessage produces WM_CHAR messages only for keys that are mapped to ASCII characters by the keyboard driver.

If applications process virtual-key messages for some other purpose, they should not call TranslateMessage. For instance, an application should not call TranslateMessage if the TranslateAccelerator function returns a nonzero value.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.427 UnhookWindowsHook

The **function is obsolete. It is provided only for compatibility with 16-bit versions of Windows. Win32-based applications should use the UnhookWindowsHookEx function.**

See Also

Hooks Overview, Hook Functions

3.428 UnhookWindowsHookEx

The UnhookWindowsHookEx function removes a hook procedure installed in a hook chain by the SetWindowsHookEx function.

```
UnhookWindowsHookEx: procedure
(
    hhk         :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__UnhookWindowsHookEx@4" );
```

Parameters

hhk

[in] Handle to the hook to be removed. This parameter is a hook handle obtained by a previous call to SetWindowsHookEx.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

The hook procedure can be in the state of being called by another thread even after UnhookWindowsHookEx returns. If the hook procedure is not being called concurrently, the hook procedure is removed immediately before UnhookWindowsHookEx returns.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.429 UnionRect

The UnionRect function creates the union of two rectangles. The union is the smallest rectangle that contains both source rectangles.

```
UnionRect: procedure
(
    var lprcDst     :RECT;
    var lprcSrc1    :RECT;
    var lprcSrc2    :RECT
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__UnionRect@12" );
```

Parameters

lprcDst

[out] Pointer to the RECT structure that will receive a rectangle containing the rectangles pointed to by the lprcSrc1 and lprcSrc2 parameters.

lprcSrc1

[in] Pointer to the RECT structure that contains the first source rectangle.

lprcSrc2

[in] Pointer to the RECT structure that contains the second source rectangle.

Return Values

If the specified structure contains a nonempty rectangle, the return value is nonzero.

If the specified structure does not contain a nonempty rectangle, the return value is zero.

Windows NT/ 2000: To get extended error information, call GetLastError.

Remarks

The system ignores the dimensions of an empty rectangle — that is, a rectangle in which all coordinates are set to zero, so that it has no height or no width.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.430 UnloadKeyboardLayout

The UnloadKeyboardLayout function unloads an input locale identifier (formerly called a keyboard layout).

```
UnloadKeyboardLayout: procedure
(
    hkl         :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__UnloadKeyboardLayout@4" );
```

Parameters

hkl

[in] Input locale identifier to unload.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. The function can fail for the following reasons:

- An invalid input locale identifier was passed.
- The input locale identifier was preloaded.
- The input locale identifier is in use.

To get extended error information, call GetLastError.

Remarks

The input locale identifier is a broader concept than a keyboard layout, since it can also encompass a speech-to-text converter, an IME, or any other form of input.

Windows 95: UnloadKeyboardLayout cannot unload the system default input locale identifier. This ensures that an appropriate character set is always available for the user to type commands for the shell or names for the file system.

Windows NT/2000: UnloadKeyboardLayout can unload the system default input locale identifier.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.431 UnpackDDElParam

The UnpackDDElParam function unpacks a DDE lParam value received from a posted DDE message.

```
UnpackDDElParam: procedure
(
        msg     :dword;
        _lParam :dword;
    var puiLo   :dword;
    var puiHi   :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__UnpackDDElParam@16" );
```

Parameters

msg

[in] Specifies the posted DDE message.

lParam

[in] Specifies the lParam parameter of the posted DDE message that was received. The application must free the memory object specified by the lParam parameter by calling the FreeDDElParam function.

puiLo

[out] Pointer to a variable that receives the low-order word of lParam.

puiHi

[out] Pointer to a variable that receives the high-order word of lParam.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Remarks

PackDDElParam eases the porting of 16-bit DDE applications to 32-bit DDE applications.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.432 UnregisterClass

The UnregisterClass function unregisters a window class, freeing the memory required for the class.

```
UnregisterClass: procedure
(
    lpClassName :string;
    hInstance   :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__UnregisterClassA@8" );
```

Parameters

lpClassName

[in] Pointer to a null-terminated string or a class atom. If lpClassName is a string, it specifies the window class name. This class name must have been registered by a previous call to the RegisterClass or RegisterClassEx function. System classes, such as dialog box controls, cannot be unregistered.

If this parameter is an atom, it must be a class atom created by a previous call to the RegisterClass or RegisterClassEx function. The atom must be in the low-order word of lpClassName; the high-order word must be zero.

hInstance

[in] Handle to the instance of the module that created the class.

Return Values

If the function succeeds, the return value is nonzero.

If the class could not be found or if a window still exists that was created with the class, the return value is zero. To get extended error information, call GetLastError.

Remarks

Before calling this function, an application must destroy all windows created with the specified class.

All window classes that an application registers are unregistered when it terminates.
Class atoms are special atoms returned only by RegisterClass and RegisterClassEx.
Windows 95: All window classes registered by a .dll are unregistered when the .dll is unloaded.
Windows NT/2000: No window classes registered by a .dll registers are unregistered when the .dll is unloaded.
Requirements
Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.433 UnregisterDeviceNotification

The UnregisterDeviceNotification function closes the specified device notification handle.

```
UnregisterDeviceNotification: procedure
(
    Handle      :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__UnregisterDeviceNotification@4" );
```

Parameters
Handle
[in] Device notification handle returned by the RegisterDeviceNotification function.
Return Values
If the function succeeds, the return value is nonzero.
If the function fails, the return value is zero. To get extended error information, call GetLastError.
Requirements
Windows NT/2000: Requires Windows 2000 or later.

Windows 95/98: Requires Windows 98 or later.

3.434 UnregisterHotKey

The UnregisterHotKey function frees a hot key previously registered by the calling thread.

```
UnregisterHotKey: procedure
(
    hWnd    :dword;
    id      :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__UnregisterHotKey@8" );
```

Parameters
hWnd
[in] Handle to the window associated with the hot key to be freed. This parameter should be NULL if the hot

key is not associated with a window.

id

[in] Specifies the identifier of the hot key to be freed.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.435 UpdateWindow

The UpdateWindow function updates the client area of the specified window by sending a WM_PAINT message to the window if the window's update region is not empty. The function sends a WM_PAINT message directly to the window procedure of the specified window, bypassing the application queue. If the update region is empty, no message is sent.

```
UpdateWindow: procedure
(
    hWnd    :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__UpdateWindow@4" );
```

Parameters

hWnd

[in] Handle to the window to be updated.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Windows NT/ 2000: To get extended error information, call GetLastError.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.436 ValidateRect

The ValidateRect function validates the client area within a rectangle by removing the rectangle from the update region of the specified window.

```
ValidateRect: procedure
(
```

```
        hWnd        :dword;
    var lpRect      :RECT
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__ValidateRect@8" );
```

Parameters

hWnd

[in] Handle to the window whose update region is to be modified. If this parameter is NULL, the system invalidates and redraws all windows and sends the WM_ERASEBKGND and WM_NCPAINT messages to the window procedure before the function returns.

lpRect

[in] Pointer to a RECT structure that contains the client coordinates of the rectangle to be removed from the update region. If this parameter is NULL, the entire client area is removed.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Windows NT/ 2000: To get extended error information, call GetLastError.

Remarks

The BeginPaint function automatically validates the entire client area. Neither the ValidateRect nor ValidateRgn function should be called if a portion of the update region must be validated before the next WM_PAINT message is generated.

The system continues to generate WM_PAINT messages until the current update region is validated.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.437 ValidateRgn

The ValidateRgn function validates the client area within a region by removing the region from the current update region of the specified window.

```
ValidateRgn: procedure
(
    hWnd    :dword;
    hRgn    :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__ValidateRgn@8" );
```

Parameters

hWnd

[in] Handle to the window whose update region is to be modified.

hRgn

[in] Handle to a region that defines the area to be removed from the update region. If this parameter is NULL, the entire client area is removed.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Windows NT/ 2000: To get extended error information, call GetLastError.

Remarks

The specified region must have been created by a region function. The region coordinates are assumed to be client coordinates.

The BeginPaint function automatically validates the entire client area. Neither the ValidateRect nor ValidateRgn function should be called if a portion of the update region must be validated before the next WM_PAINT message is generated.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.438 VkKeyScan

The VkKeyScan function translates a character to the corresponding virtual-key code and shift state for the current keyboard.

This function has been superseded by the VkKeyScanEx function. You can still use VkKeyScan, however, if you do not need to specify a keyboard layout.

```
VkKeyScan: procedure
(
    _ch     :char
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__VkKeyScanA@4" );
```

Parameters

ch

[in] Specifies the character to be translated into a virtual-key code.

Return Values

If the function succeeds, the low-order byte of the return value contains the virtual-key code and the high-order byte contains the shift state, which can be a combination of the following flag bits.

Bit	Meaning
1	Either SHIFT key is pressed.
2	Either CTRL key is pressed.
4	Either ALT key is pressed.
8	The Hankaku key is pressed
16	Reserved (defined by the keyboard layout driver).
32	Reserved (defined by the keyboard layout driver).

If the function finds no key that translates to the passed character code, both the low-order and high-order bytes contain –1.

Remarks

For keyboard layouts that use the right-hand ALT key as a shift key (for example, the French keyboard layout), the shift state is represented by the value 6, because the right-hand ALT key is converted internally into CTRL+ALT.

Translations for the numeric keypad (VK_NUMPAD0 through VK_DIVIDE) are ignored. This function is intended to translate characters into keystrokes from the main keyboard section only. For example, the character "7" is translated into VK_7, not VK_NUMPAD7.

VkKeyScan is used by applications that send characters by using the WM_KEYUP and WM_KEYDOWN messages.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.439 VkKeyScanEx

The VkKeyScanEx function translates a character to the corresponding virtual-key code and shift state. The function translates the character using the input language and physical keyboard layout identified by the input locale identifier.

```
VkKeyScanEx: procedure
(
    _ch     :char;
    dwhkl   :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__VkKeyScanExA@8" );
```

Parameters

ch

[in] Specifies the character to be translated into a virtual-key code.

dwhkl

[in] Input locale identifier used to translate the character. This parameter can be any input locale identifier previously returned by the LoadKeyboardLayout function.

Return Values

If the function succeeds, the low-order byte of the return value contains the virtual-key code and the high-order byte contains the shift state, which can be a combination of the following flag bits.

Bit	Meaning
1	Either SHIFT key is pressed.
2	Either CTRL key is pressed.
4	Either ALT key is pressed.
8	The Hankaku key is pressed
16	Reserved (defined by the keyboard layout driver).
32	Reserved (defined by the keyboard layout driver).

If the function finds no key that translates to the passed character code, both the low-order and high-order bytes

contain –1.

Remarks

The input locale identifier is a broader concept than a keyboard layout, since it can also encompass a speech-to-text converter, an IME, or any other form of input.

For keyboard layouts that use the right-hand ALT key as a shift key (for example, the French keyboard layout), the shift state is represented by the value 6, because the right-hand ALT key is converted internally into CTRL+ALT.

Translations for the numeric keypad (VK_NUMPAD0 through VK_DIVIDE) are ignored. This function is intended to translate characters into keystrokes from the main keyboard section only. For example, the character "7" is translated into VK_7, not VK_NUMPAD7.

VkKeyScanEx is used by applications that send characters by using the WM_KEYUP and WM_KEYDOWN messages.

Requirements

Windows NT/2000: Requires Windows NT 4.0 or later.

Windows 95/98: Requires Windows 95 or later.

3.440 WaitForInputIdle

The WaitForInputIdle function waits until the specified process is waiting for user input with no input pending, or until the time-out interval has elapsed.

```
WaitForInputIdle: procedure
(
    hProcess        :dword;
    dwMilliseconds  :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__WaitForInputIdle@8" );
```

Parameters

hProcess

[in] Handle to the process. If this process is a console application or does not have a message queue, WaitForInputIdle returns immediately.

dwMilliseconds

[in] Specifies the time-out interval, in milliseconds. If dwMilliseconds is INFINITE, the function does not return until the process is idle.

Return Values

The following table shows the possible return values:

Value	Meaning
0	The wait was satisfied successfully.
WAIT_TIMEOUT	The wait was terminated because the time-out interval elapsed.
WAIT_FAILED	An error occurred. To get extended error information, use the GetLastError function.

Remarks

The WaitForInputIdle function enables a thread to suspend its execution until the specified process has finished its initialization and is waiting for user input with no input pending. This can be useful for synchronizing a parent process and a newly created child process. When a parent process creates a child process, the CreateProcess function returns without waiting for the child process to finish its initialization. Before trying to communicate with the child process, the parent process can use WaitForInputIdle to determine when the child's initialization has been completed. For example, the parent process should use WaitForInputIdle before trying to find a window associated with the child process.

The WaitForInputIdle function can be used at any time, not just during application startup.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.441 WaitMessage

The WaitMessage function yields control to other threads when a thread has no other messages in its message queue. The WaitMessage function suspends the thread and does not return until a new message is placed in the thread's message queue.

```
WaitMessage: procedure;
    @stdcall;
    @returns( "eax" );
    @external( "__imp__WaitMessage@0" );
```

Parameters

This function has no parameters.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

Note that WaitMessage does not return if there is unread input in the message queue after the thread has called a function to check the queue. This is because functions such as PeekMessage, GetMessage, GetQueueStatus, WaitMessage, MsgWaitForMultipleObjects, and MsgWaitForMultipleObjectsEx check the queue and then change the state information for the queue so that the input is no longer considered new. A subsequent call to WaitMessage will not return until new input of the specified type arrives. The existing unread input (received prior to the last time the thread checked the queue) is ignored.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.442 WinHelp

Starts Microsoft® Windows® Help (Winhelp.exe) and passes additional data that indicates the nature of the help

requested by the application.

```
WinHelp: procedure
(
    hWndMain    :dword;
    lpszHelp    :string;
    uCommand    :dword;
    dwData      :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__WinHelpA@16" );
```

Parameters

hWndMain

Handle to the window requesting help. The WinHelp function uses this handle to keep track of which applications have requested help. If the uCommand parameter specifies HELP_CONTEXTMENU or HELP_WM_HELP, hWndMain identifies the control requesting help.

lpszHelp

Address of a null-terminated string containing the path, if necessary, and the name of the help file that WinHelp is to display.

The file name can be followed by an angle bracket (>) and the name of a secondary window if the topic is to be displayed in a secondary window rather than in the primary window. You must define the name of the secondary window in the [WINDOWS] section of the help project (.hpj) file.

uCommand

Type of help requested. For a list of possible values and how they affect the value to place in the dwData parameter, see the Remarks section.

dwData

Additional data. The value used depends on the value of the uCommand parameter. For a list of possible dwData values, see the Remarks section.

Return Values

Returns nonzero if successful, or zero otherwise. To get extended error information, call GetLastError.

Remarks

Before closing the window that requested help, the application must call WinHelp with the uCommand parameter set to HELP_QUIT. Until all applications have done this, Windows Help will not terminate. Note that calling Windows Help with the HELP_QUIT command is not necessary if you used the HELP_CONTEXTPOPUP command to start Windows Help.

The following table shows the possible values for the uCommand parameter and the corresponding formats of the dwData parameter.

uCommand	Action	*dwData*
HELP_COMMAND	Executes a help macro or macro string.	Address of a string that specifies the name of the help macro(s) to run. If the string specifies multiple macro names, the names must be separated by semicolons. You must use the short form of the macro name for some macros because Windows Help does not support the long name.

HELP_CONTENTS	Displays the topic specified by the Contents option in the [OPTIONS] section of the .hpj file. This command is for backward compatibility. New applications should provide a .cnt file and use the HELP_FINDER command.	Ignored; set to 0.
HELP_CONTEXT	Displays the topic identified by the specified context identifier defined in the [MAP] section of the .hpj file.	Contains the context identifier for the topic.
HELP_CONTEXTMENU	Displays the Help menu for the selected window, then displays the topic for the selected control in a pop-up window.	Address of an array of double word pairs. The first double word in each pair is the control identifier, and the second is the context identifier for the topic. The array must be terminated by a pair of zeros {0,0}. If you do not want to add help to a particular control, set its context identifier to -1.
HELP_CONTEXTPOPUP	Displays the topic identified by the specified context identifier defined in the [MAP] section of the .hpj file in a pop-up window.	Contains the context identifier for a topic.
HELP_FINDER	Displays the Help Topics dialog box.	Ignored; set to 0.
HELP_FORCEFILE	Ensures that Windows Help is displaying the correct help file. If the incorrect help file is being displayed, Windows Help opens the correct one; otherwise, there is no action.	Ignored; set to 0.
HELP_HELPONHELP	Displays help on how to use Windows Help, if the Winhlp32.hlp file is available.	Ignored; set to 0.
HELP_INDEX	Displays the topic specified by the Contents option in the [OPTIONS] section of the .hpj file. This command is for backward compatibility. New applications should use the HELP_FINDER command.	Ignored; set to 0.
HELP_KEY	Displays the topic in the keyword table that matches the specified keyword, if there is an exact match. If there is more than one match, displays the Index with the topics listed in the Topics Found list box.	Address of a keyword string. Multiple keywords must be separated by semicolons.
HELP_MULTIKEY	Displays the topic specified by a keyword in an alternative keyword table.	Address of a MULTIKEYHELP structure that specifies a table footnote character and a keyword.
HELP_PARTIALKEY	Displays the topic in the keyword table that matches the specified keyword, if there is an exact match. If there is more than one match, displays the Topics Found dialog box. To display the index without passing a keyword, use a pointer to an empty string.	Address of a keyword string. Multiple keywords must be separated by semicolons.
HELP_QUIT	Informs Windows Help that it is no longer needed. If no other applications have asked for help, Windows closes Windows Help.	Ignored; set to 0.
HELP_SETCONTENTS	Specifies the Contents topic. Windows Help displays this topic when the user clicks the Contents button if the help file does not have an associated .cnt file.	Contains the context identifier for the Contents topic.

HELP_SETPOPUP_POS	Sets the position of the subsequent pop-up window.	Contains the position data. Use MAKELONG to concatenate the horizontal and vertical coordinates into a single value. The pop-up window is positioned as if the mouse cursor were at the specified point when the pop-up window was invoked.
HELP_SETWINPOS	Displays the Windows Help window, if it is minimized or in memory, and sets its size and position as specified.	Address of a HELPWININFO structure that specifies the size and position of either a primary or secondary help window.
HELP_TCARD	Indicates that a command is for a training card instance of Windows Help. Combine this command with other commands using the bitwise OR operator.	Depends on the command with which this command is combined.
HELP_WM_HELP	Displays the topic for the control identified by the hWndMain parameter in a pop-up window.	Address of an array of double word pairs. The first double word in each pair is a control identifier, and the second is a context identifier for a topic. The array must be terminated by a pair of zeros {0,0}. If you do not want to add help to a particular control, set its context identifier to -1.

See Also

HELPWININFO, MULTIKEYHELP

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.443 WindowFromDC

The WindowFromDC function returns a handle to the window associated with the specified display device context (DC). Output functions that use the specified device context draw into this window.

```
WindowFromDC: procedure
(
    hDC     :dword
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__WindowFromDC@4" );
```

Parameters

hDC

[in] Handle to the device context from which a handle for the associated window is to be retrieved.

Return Values

The return value is a handle to the window associated with the specified DC. If no window is associated with the specified DC, the return value is NULL.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

3.444 WindowFromPoint

The WindowFromPoint function retrieves a handle to the window that contains the specified point.

```
WindowFromPoint: procedure
(
    _Point      :POINT
);
    @stdcall;
    @returns( "eax" );
    @external( "__imp__WindowFromPoint@8" );
```

Parameters

Point

[in] Specifies a POINT structure that defines the point to be checked.

Return Values

The return value is a handle to the window that contains the point. If no window exists at the given point, the return value is NULL. If the point is over a static text control, the return value is a handle to the window under the static text control.

Remarks

The WindowFromPoint function does not retrieve a handle to a hidden or disabled window, even if the point is within the window. An application should use the ChildWindowFromPoint function for a nonrestrictive search.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.